Unit 1 - Data Structures

Asymptotic Notations → Define all three → egs → Draw graph

Recurrence Relation

Data structure? & classification & operations

Unit 2 - Stack

Algo for push & pop

Graphical representation diagram

Polish Notations / Stack Traversal

What is recursion? Justify with eg (Tower of Hanoi / any recursive problem)

Tower of Hanoi → define algo, eg, graphical representation, tracing (opt)

Unit 3 - Queue & Linked List

Insertion (enqueue) & deletion (dequeue) in queue → algo, diagram

Linked list & types → draw graphical structure → createFirst, addNode

Difference b/w union & structure

Unit 4 - Tree

Define binary tree, BST, skewed tree

~~[illegible]~~ AVL, B, B$^+$, Red black → short notes, numerical, complexity, algo, definition

Difference b/w B & B$^+$ tree

Tree Traversal → algo, egs, numerical

Unit 5 - Sorting & Searching

Merge, Quick, Heap (imp) → algo, defn, complexity, eg, ~~[illegible]~~ applications [Construct min heap → first make max Heap then min Heap the complexity]

Difference b/w sorting & searching

Types of searching → linear, binary → which is better, complexity, numerical

Graph

Representation, definition, traversal (DFS, BFS) → algo, egs, complexity

Spanning tree, min span tree → technique (Kruskal, prim's) → numerical

Data Structures

Data structures are logical entities used to organise the data in the system

Functions that can be performed on data structures :-

1> Create
2> Insert
3> Delete
4> Modify / Update
5> Searching
6> Sorting
7> Merging

Data :- unorganized Raw facts & figures which doesn't have any meaning

Information :- * Processed data is called information, it has meaningful format.

* ~~By pr.~~ Processing the data means implementing ~~implet~~ data structures.

⇒ Data structure is the way of organising data in the system, apart from that it tells us how efficiently we can fetch, store & process the data. It gives us the logical or it is used to represent the relationship between the data elements.

Application of data structures :-

i> Big data → framework : Hadoop ~~adobe~~, hive, spark, strom
ii> AIML
iii> DBMS

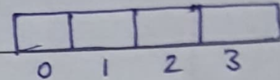
Java → unicode
generally ASCII encoding

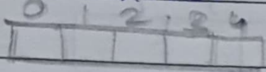
Types of Data Structures

- Primitive
  - Pointers
  - int
    - byte
    - short → ? ~~byte~~
    - int → 4 byte 2^{31} to -2^{31}
    - long
  - char
  - float
- Non-primitive
  - Linear [single level storage, same priority for all elements]
    - → Arrays [sorted & sequencial with same datatype] (homogeneous)
    - ADT Abstract Datatype
      - → Linked list → [Dynamic memory allocation]
      - → Stack
      - → Queue
  - Nonlinear [random storage, any level]
    - → Graph
    - → Trees

Arrays

* Arrays are non-primitive linear data structure
* It uses continuous memory allocation.





Syntax :- returnType arrname[size];

eg :- int a[5]; } Declaration (garbage values are assigned)
float b[10];

* Array can hold (n) number of elements and index goes from 0 to n-1.

* eg → int a[5] = {18, 25, 1, 7, 3} or int a[ ] = {18, 25, 1, 7, 3}
//definition

0	1	2	3	4	→ index (For user)
18	25	1	7	3	
100	104	108	112	116	→ address

4 bytes for int

→ int a [5] = {18, 25} /*other three places are assigned garbage values */

→ int a[3] = {18, 25, 1, 7} //

• The above method is called static declaration /initialization

* Retrieving values from array :- [Indexing is used]
a [5] = {18, 25, 1, 7, 3}

⇒ a [0] = 18
⇒ a [4] = 3

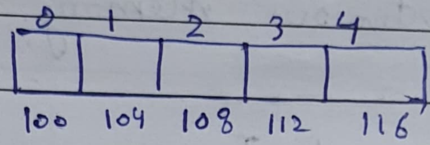
To display all values, we use for loop
⇒ for (i=0 ; i<n ; i++) { a [i]; }

// <, >, <=, >= → relational operator , ++, -- → unary operator

* internal working for finding address from index

Address of block = Base Address of array + (index * size of one block)

eg → int a[5];



a[4] = 100 + (4 × 4) = 100 + 16 = 116

a[4] ↓ index

116 ↓ address

Printf & scanf

Syntax → printf ("message and format specifier", argument list)

scanf ("only format specifier", address list)

address list ↓ ampersand (&) [address of]

Lifecycle of a program:-

Source Code (.c) → preprocessor directives → expansion of Source code → Compiler

↓

Linker ← Object Code (.obj) ← Assembler ← Assembly Code (.s) (intermediate lang)

↓

executable code (.exe) → Loader (loads on Primary memory) → Run on main memory → output on console

---

Array → Array is a data structure which is used to store the elements of same type in continous memory location that can be individually referenced by using an index number which will be the unique identifier in case of arrays.

⇒ Properties of array:-

i) The array index starts from the number 0.

ii) Its range will be 0 to n-1.

iii) It can hold any type of elements like floating type array, character type array, integer type array, string type array.

⇒ Types of arrays:-

1) 1-Dimensional array

2) 2-Dimensional array

3) 3-Dimensional / Multi dimensional array

→ Application of arrays :-

i) It is used in CPU scheduling in operating system to store the multiple processes.

ii) It can be used to store the contact records in phone book software.

iii) The files that we attach with the mail is an example of array.

⇒ Syntax of array :-

→ To declare an array

returntype arrName [arrSize];

eg→ int a[15];

→ To define / initialize an array.

return type arrName [arr Size] = {arr Elements};

eg→ int a[5] = {10, 20, 30, 40, 50};

→ To initialize all memory blocks by 0.

returntype arrName [arr Size] = {0};

eg→ int a[5] = {0};

⇒ Calculation of address in 1-D array :-

$$\boxed{\text{Address of block} = \text{Base address of array} + [\text{index} \times \text{size of one block}]}$$

Q) WAP to take the ~~starti~~ static input from the user in 1-D array.

Sol. int main ( ) { int a[3] = {10, 5, 18};

printf ("%d", a[0]);

printf ("%d", a[1]);

printf ("%d", a[2]); }

Method 1
[Compile-time]
initialization

⇒ 10
5
18

a =

10	5	18
0	1	2

Method 2 If we have a larger dataset then we will use the for loop:

```
int main() { int a[6] = {10, 5, 18, 19};
        for (int i=0 ; i<6 ; i++)
              { printf ("%d", a[i]); }
          }
```

⇒
10
5
18
19

0	1	2	3
10	5	18	19

a

Q) Take dynamic input from user during runtime.

Sol

```
int main() { int a[5];
```

Storing data

```
   printf ("Enter elements in an array");
   for (int i=0; i<5, i++) { scanf ("%d", &a[i]); }
```

displaying data

```
   printf ("Elements of stored Array are");
   for (i=0; i ≤ 4; i++) { printf ("%d", a[i]); }
          }
```

Q> WAP to the value of array at runtime

Ans
```
int main ( ) { int sizeOfArray;
        printf ("Enter size of array: ");
        scanf ("%d", &sizeOfArray);
    int arr[sizeOfArray];
    for (int i=0 ; i<sizeOfArray; i++)
{ printf ("Enter value at %d index: ");
   scanf ("%d", &arr[i]); }
  int index, value;
 printf ("Enter the index between 0 to %d to change
        its value: ", sizeOfArray);
 scanf (" %d", & index);
 printf ("Enter new value: ");
 scanf ("%d", & value);
  arr[index] = value;
 printf ("Updated Array is : ");

 for (i=0 ; i<sizeOfArray ; i++)
  { printf ("%d", arr[i]); }

        }
```

Q> WAP a program to calculate the sum of n numbers stored in an array.

Sol
```
 int main ( ) {   int arr[5] = {1, 2, 3, 4, 5};
        int sum=0;
        for (int i=0; i<5; i++) {
            sum += arr[i]; }
printf ("%d", sum); }
```

Q> WAP to copy one array into another array & the input will be taken during runtime.

Sol

```
int main ( ) { int sizeofarray ; printf ("Enter size of first array : ");
                scanf ("%d" , &sizeofarray ); int arr1 [sizeofarray];
for (int i=0 ; i<sizeofarray ; i++)
    { printf ("Enter value at %d index :  ");
      scanf ("%d", &arr1[i]) ; }
  int arr2 [sizeofarray] ; for (i=0 ; i<sizeofarray ; i++){
  { arr1[i] = arr2[i] ;} printf ("Second array is : ");
  for (i=0 ; i<sizeofarray ; i++){ printf ("%d", arr2[i]) ; }}
```

2-Dimensional Array

Array within array is known as 2-D array, it can viewed as the combination of rows & columns i.e matrix form.

Note: In 2-D array column field is mandatory.

Employee table

~~For storing~~ rows

ID	Name	Design	Salary

Columns

2D array syntax (static initialisation)

returntype arrname [no. of rows] [no. of columns]

eg→ int b [5] [4] ;

int b [2] [2] = { 10, 18, 25, 16 } ;

	1	2
1	b_{11}	b_{12}
2	b_{21}	b_{22}

10	18
25	16

Syntax 2 : int b [2] [2] = { {10, 18} , { 25, 16} };

int b [2][4] = { {10, 18, 15; -67}, {-58, 67, -82, 108} };

int a [ ] = {10, 5, 3} ; [valid in 1-D array]

// In two-dimensional array column field is mandatory.

eg : int c[ ] [2] = {{10, 2}, {3, 8} } ; [valid]
int c[ ] [ ] = { {10, 2}, {3, 8} } ; [not valid]
int c [2] [ ] = {{10, 2}, {3, 8} } ; [not valid]

⇒ Dynamic initialisation:

```
int main () { int marks [10] [10];
    for (int i = 0 ; i ≤ 9 ; i++) {
        for (int j = 0 ; j ≤ 9 ; j++) {
            scanf (" % d" , & marks [i] [j]);
        }}
    for (int i = 0 ; i ≤ 9 ; i++) {
        for (int j = 0 ; j ≤ 9 ; j ++) {
            printf (" %d", marks [i] [j]);
        }} }
```

⇒ Multidimensional array:

int a [2] [3] [2] = { { {10, 12}, {24, -67}, {-18, 17} },
{ { 15, 76}, {18, 5} , {6, 7} } }

Instruction

variable declaration
int a, float salary

Decision ctrl
if - else

ctrl instructions
for, while, do-while

Complexity → Efficiency of the program

→ Space Complexity
→ Time Complexity

Three Cases :- (to describe efficiency of program

	Asymptotic Notation
1) Best Case	Ω
2) Average Case	Θ
3) Worst Case	Big O

Asymptotic Notation

Big O - Let us suppose there are two functions $f(n)$ and $g(n)$ then $\boxed{f(n) = O\, g(n)}$ if and only if $\boxed{f(n) \le c \cdot g(n)}$ for all $n \ge n_0$ and for any constant c

there exist two possible const c & n_0 such that

eg → consider a linear function $f(n) = 3n+5$

$\because f(n) \le c \cdot g(n)$ and $n \ge n_0$

$3n+5 \le 3n+n$ assuming $n_0 = 5$

$3n+5 \le 4n$ $\therefore n = 5$

$3n+5 \le 4\, g(n)$

$\therefore \boxed{n_0 = 5, \; c = 4}$

→ $f(n) = 10n+16$

$\because n \ge n_0$ $\therefore$ assuming $n_0 = 16 \Rightarrow n = 16$

$\because F(n) \le c\,(g(n))$

$f(n) \le 10n + n$

$f(n) \le 11n$

$f(n) \le 11 \cdot g(n)$ $\therefore \boxed{n_0 = 16, \; c = 11}$

$\rightarrow$ $f(n^2) = 7n^2 + 2n + 1$

$f(n^2) = O\,g(n^2)$

$\Rightarrow$ $f(n^2) \leq c \cdot g(n^2) \;\forall\; n \geq n_0$

Assuming $n_0 \geq 1$

$\therefore n \geq 1$

$\therefore 7n^2 + 2n + 1 \leq 7n^2 + 2n + n$

$7n^2 + 2n + 1 \leq 7n^2 + 3n$ ~~Assuming $3n \geq n^2$~~

$\therefore 7n^2 + 2n + 1 \leq 7n^2 + n^2$ i.e $n_0 \geq n^2$

$7n^2 + 2n + 1 \leq 8n^2$ Replacing $3n$ by n^2

$f(n^2) \leq 8 \cdot g(n^2)$

$\therefore \boxed{c = 8}$, $n_0 \geq n^2$

$\therefore \boxed{n_0 = 1}$

$\rightarrow$ $f(n^3) = 11n^3 + 8n^2 + 10$

$f(n^3) = O\,g(n^3)$

$\Rightarrow$ $f(n^2) \leq c \cdot g(n^2) \;\forall\; n \geq n_0$

Replacing 10 by n^2

$\therefore 11n^3 + 8n^2 + 10 \leq 11n^3 + 8n^2 + n^2$

$11n^3 + 8n^2 + 10 \leq 11n^3 + 9n^2$

Replacing $9n^2$ by n^3

$11n^3 + 8n^2 + 10 \leq 11n^3 + n^3$

$11n^3 + 8n^2 + 10 \leq 12n^3$

$\therefore c = 12$, $n = 9$

Time Complexity

There are two methods to find out the time complexity for any program:

1) Table method

(Count method) 2) Global variable count method, initialized by 0 always for the instruction which will get executed.

Let us consider the example for adding the array elements having (n) number of elements.

```
          global → count = 0
    Algo Sum (A, n) {
              sum := 0 ;
              ↳ count = count + 1;

           for (i=0 ; i<n; i++) {
                  ↳ Count = count + 1 ;
                Sum := sum + A[i];
                  ↳ Count = count + 1;
           }
           count := count + 1 ;   // for termination cond of loop

           return sum;
             ↳ count := count + 1 ;
    }
```

$\therefore$ Count $= 3 + 2n$

$\boxed{\text{Time complexity} = O(n)}$ $\because$ 3 & 2 are constant

Q> WAP for ~~matrix addition~~ adding elements of matrix and find its time complexity.

Ans

```
Algo MatAdd (A, B, n, m){
    C[n][m];
    ↳ count := count +1;

    for (i=0 ; i<n ; i++){
        for (j=0 ; j<m ; j++) {
            C[i][j] = A[i][j] + B[i][j];

        }
    }

return C; }
```

```
Algo Sum(A, r, c) {
    Sum:=0 → c=c+1; // sum

    for (i=0 ; i<r ; i++) {  c=c+1; // outer loop
        for (j=0 ; j<c ; j++) { c=c+1; // inner loop

            sum := sum + A[i][j] + B[i][j];
                  c=c+1; // sum
        } c=c+1; // terminate inner loop
    } c=c+1; // terminate outer loop
~~retun~~ .
    return sum; c=c+1; // return
}
```

$c = 3 + 2rc + 2r$

Time Complexity = $O(rc)$

While loop → n times execution
For loop → n+1 times execution

Date
Page

Table method to calculate time complexity.

→ s/e → steps per execution
freq → frequency
Total → s/e * freq

(adding n elements of array)

instruction	s/e	freq	Total
Algo Sum	1	0	0
Sum=0	1	1	1
for loop	1	n+1	n+1
Sum = sum + A[i]	1	n	n
return sum	1	1	1
			2n+3

∴ Time complexity = O(n)

Searching

→ linear search / sequencial search
* starts from index 0.
* each element is compared to key element until the last element and then it is seen if key element exists in array or not.

eg →

	0	1	2	3	4	5
a =	10	8	5	9	15	25

if key = 10 → element found at a[0] ⇒ Best case O(1)
if key = 15 → element found at a[4] = Avg case O(n)
if key = 25 → element found at a[5] = worst case O(n)

Searching
* Searching is the process of fetching a specific element in a collection of elements.
* The elements can be collected in any datastructure.
* If you find the element in a list, the process is considered as successful, & it returns the location of that element, & in contrast, if you do not find

the element it seems the search unsuccessful.

There are two prominent strategies to find a specific element in a list:-

1> Linear Search / Sequential search.
2> Binary Search.

linear search - * It is the most basic search ~~teg~~ technique as in this type of searching, you go through the list & try to fetch a match for a single element.

* If you find a match then the address of the matching target element is returned.

* On the other hand, if the element is not found then it returns a null value.

⇒ Best Case complexity :- * The element being searched could be found in the first position.

* In this case, the search ends with a single successful comparison and thus the time complexity will be O(1).

⇒ Worst Case complexity :- * The element being searched maybe at the last position in the array or it does not exist in the array.

* In the first case the search ~~successed~~ succeed ~~successed~~.

In the next case the search fails after n comparison. Thus the worst ~~cost~~ case time complexity will be $O(n)$.

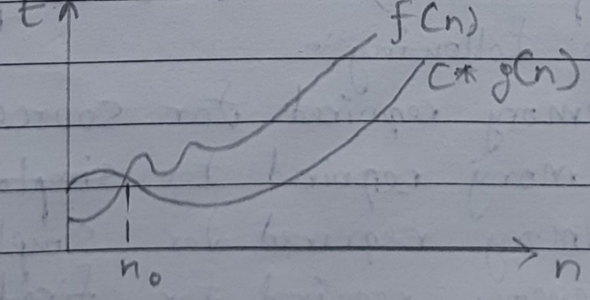
⇒ Average Case Complexity :- * when the element to be searched is in the middle of array then time complexity will be $O(n)$.

Space complexity of linear search
The linear search will take the space of $O(n)$ for n number of elements in array.

Application of linear search :- (Drawbacks)

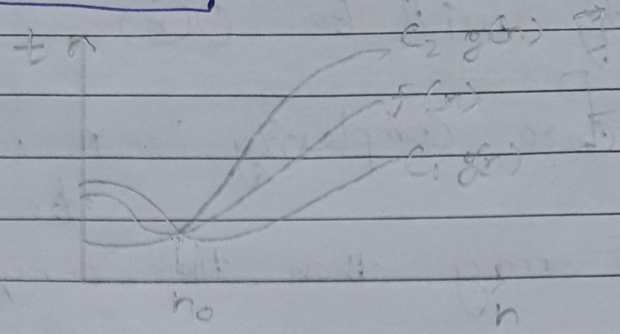
i) When the list will contain less amount of elements.
ii) When the ~~fit~~ searching for a single element in an unordered array.

Omega notation (Ω) [Best case] - Let $f(n)$ & $g(n)$ are two positive functions then $f(n)$ will be equal to $\Omega \cdot g(n)$ i.e $\boxed{f(n) = \Omega\, g(n)}$ if and only if there exist two positive constant c & n_0 such that $\boxed{f(n) \geq c \cdot g(n)}$ for all $\boxed{n \geq n_0}$ and $c > 0$ & $n \geq 1$ and this always represents the best case.



Theta notation (θ) [Average Case] - Let $f(n)$ & $g(n)$ are two positive functions such that $\boxed{f(n) = \theta\, g(n)}$ if and only if there exists three positive constant c_1, c_2 & n_0 such that

$\boxed{c_1\, g(n) \le f(n) \le c_2\, g(n)}$ for all $\boxed{n \ge n_0}$



Complexity - The complexity is defined as the overall efficiency of program. It can be measured depending on two factors :-

1> **Space Complexity**

* The space complexity of the algorithm is the overall memory required for the algorithm to its completion.
* Each algoritm when it is carried out on any computer ~~wt~~ through the equivalent program, it requires some memory for its execution.
* This space requirement maybe of two types :-

(Fixed part) i> There will be a fixed path which is ~~inpedent~~ independent of its instance characteristics (number and size of input & output parameters). This fixed part contains memory for the following :-

a> Memory required for source code.
b> Memory required for ~~simple~~ ~~var~~ constants.
c> Memory required for simple variables
d> Memory required for fixed size components used in the program.

ii> Variable part - The variable part memory is dependent upon the instance characteri-

-stics of the algorithm. The variable part memory contains memory for the following :-

a) Memory required for reference variable
b) Memory required for instance variable
c) Memory required for recursion stack

Hence, the space complexity can be defined as

$$\boxed{S(P) = c + S_p}$$

where c = fixed part which does not depend on the instance characteristics

and S_p = variable part which depends on instance characteristics.

2) Time Complexity

* The time complexity of an algorithm is the time required for algorithm to its completion.
* The total time required may be the sum of compile time & runtime.
* The compile time for an algorithm may again be assumed as constant, because once the program is compiled it can be executed any number of time without any more compilation.
* The runtime of any program may vary depending upon the number of input and time of operation performed along with the type of data on which the operation is performed.

eg → The multiplication takes more time as compared to addition for floating / double type values.

* The execution time also depends upon the hardware Configuration of the system.

There are two ways to compute the time complexity :-

i) Global variable Count : By initializing global variable count from 0 which ~~and~~ is increased by one if a single instruction is executed

// eg in previous page (algo Sum)

// first theory then algo, then application then

// advantage / disadvantage then complexity (for exams)

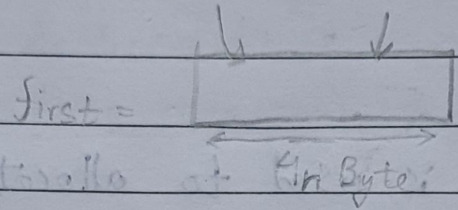
Linked List

⇒ Malloc

syntax → ~~cast~~ pointer = (cast_type *) malloc (byte_size);

eg → int *first = (int *) malloc (n * sizeof (int));
↳ garbage value

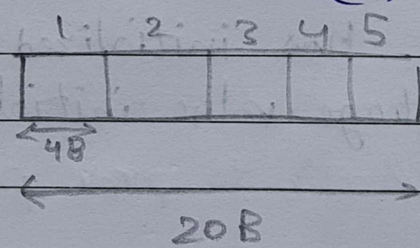
int *first = (int *) malloc (sizeof (int));
↳ 4 Byte



⇒ Calloc

Syntax → pointer = (cast_type *) calloc (size, sizeof (int));

eg → int * first = (int *) calloc (5, size of (int)) ;



⇒ Realloc

→ used after malloc or calloc

eg→ int* first = (int*) malloc (5 * size of (int));

first = realloc (first, 3* sizeof (int));

Free (first);

syntax → pointer_name = realloc (prevAllocated Pointer, new size);

Dynamic Memory Allocation

Dynamic memory allocation can be defined as a procedure in which the size of the data structure is changed during run-time..

There are some functions to achieve these tasks.

These 4 library functions are defined under the ~~library~~ <stdio.h> header file to allocate dynamic memory in C :

malloc - * malloc method is used to allocate memory dynamically and it allocates single large block of memory with the specified size.

* It returns a pointer of type void which can be cast into a pointer of any form.
* It does not initialize memory at execution time so that it has initialized each block with the default garbage value initially.

// refer to prev. page for syntax & eg

calloc - * calloc is also used to allocate memory dynamically and a specified number of blocks of specified type.

* It is similar to malloc but it as two different factors:
  i) It has two parameters as compared to malloc.
  ii) It initializes each block with a default value 0.

realloc - * It is used to dynamically change the ~~dyn~~ memory allocation of a previously allocated memory.

* In other words, if memory allocated with help of malloc or calloc is insufficient then the realloc function can be used to dynamically reallocate the memory.

* Reallocation of memory maintains the already present value & new blocks will be ~~~~ initialized with default garbage value.

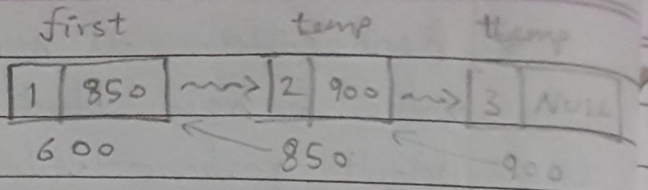
free - ~~~~This method is used to dynamically ~~det~~ deallocate the memory.
* The memory allocated using the function malloc, calloc and realloc is deallocated by using the free function.
* It helps to reduce the wastage of memory by freeing it.

Q) WAP to create singly linked list by making use of structure.

Ans

```
typedef struct SinglyLinkedList {
        int data;
        struct SinglyLinkedList *next;
    } node;

node *first, *temp, *ttemp ---- ; //global ptr

void createfirst (int val) {
    first = (node *) malloc (sizeof(node));
    first → data = val ;
    first → next = null ;
}

void addNode ( int val) {
    temp = first ;
    while (temp → next ! = Null) { temp = temp → next ; }
    ttemp = (node *) malloc (size of (node)) ;
    ttemp → data = val ;
```

```
    ttemp -> next = Null;
    temp -> next = ttemp;
}
```



Q> WAP to add a node in front of existing linked list.

Ans

```
void addNodeInFront (int val) {
        P = (node *) malloc (sizeof(node));
        P -> data = val;
        P -> next = NULL;
        P -> next = first;
        first = P;
    }
```

Q> WAP to add node to last position of linked list.

Ans

```
void addNodeInLast (int val) {
        temp = first;
    while (temp -> next != null) { temp = temp -> next; }
     ttemp = (node *) malloc (sizeof (node));
     ttemp -> data = val;
     ttemp -> next = NULL;
      temp -> next = ttemp;
    }
```

Q> WAP to swap the first & last node of linked list.

Ans

```
node *first, *temp, *ttemp, *P;

void swapNode (int a, int b) {
        temp = first;
    while (temp -> next != NULL) { ttemp = temp;
```

```
        temp = temp -> next;
    }
    // temp = te
        P = first -> next;
      temp -> next = P;
      first -> next = NULL;
      temp -> next = first;
       first = temp;
    }
```

User Defined Datatypes

The basic set of datatypes defined in C language (i.e int, char, float etc) maybe insufficient for your application. In these circumstances you can create your own datatype which are based on some standard.

There are three mechanisms to do so :-

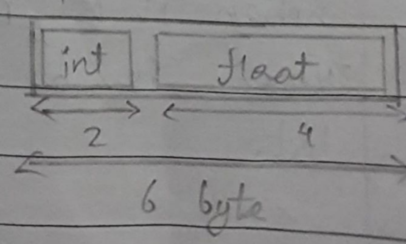
i) structure (ii) Union (iii) typedef (It is a keyword which improves the readability of code by means of abbreviating it).

Structure * It is the user defined datatype used to represent the records.

* It is a collection of different datatypes which are grouped together & each element in C structure is called a member.
* If you want to access the structure member in C, structure variable should be declared.
* Many structure variables can be declared for same structure & the memory will be allocated for each separately.

eg →

```
struct MyStruct{ int a;
                 float b; }
```



* To access a member of structure ~~pointer operator (->)~~ using pointers we use (->) operator.
~~and~~ otherwise we use (.) operator.

eg →
```
struct person { int age;
                float weight; }
int main() {
    struct person *personPtr, person1;
    personPtr= &person1;

    printf("Enter age: ");
    scanf("%d", person1.age);

    printf("Enter weight: ");
    scanf("%f", &personPtr->weight);

    printf("Displaying: \n");
    printf("Age: %d \n", personPtr->age);
    printf("Weight: %f", person1.weight);
  return 0;
}
```

Note ⇒ personPtr → age is equivalent to (*personPtr).age

Linked List

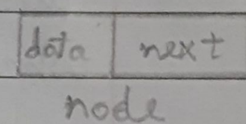
If the memory is allocated before the execution of program it is fixed and cannot be changed. We have to adopt an alternate strategy to allocate the memory only when it is required.
There is a special data structure known as linked list.

* Linked list are special list of some data element linked to one another.
* Each element is called node which has two parts :-
  i> Data part which stores information
  ii> Pointer part which stores address pointing to the next element.

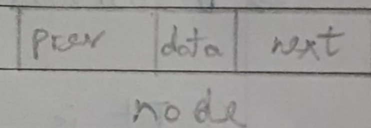
⇒ Types of linked list :

1> Singley linked list

In singley linked list, nodes have one pointer (next) pointing to the next node.

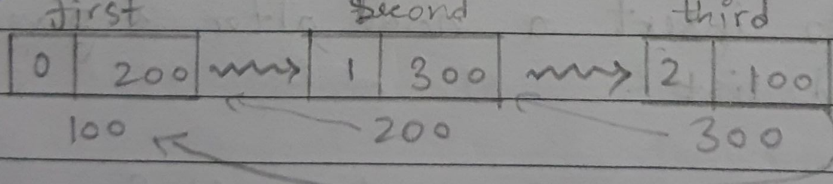


2> Doubley linked list

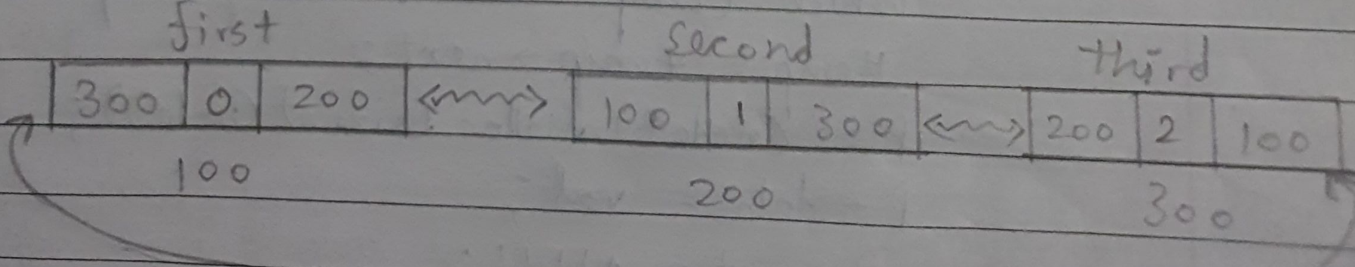


Nodes of doubley linked list have two pointers (previous & next). Previous points to the previous node address & next point to the next node address.

3> Singley Circular linked list



4> Doubley Circular linked list



⇒ Operations performed on linked list :-

i> Insertion
ii> Creation
iii> Traverse
iv> Deletion
<v> Searching
<vi> Concatenation
<v> Display

⇒ Advatages of linked list :-

i> Insertion & deletion are easier & efficient.
ii> Efficient memory utilisation. Here, memory is allocated wherever it is required and deallocated when it is no longer needed.

⇒ Disadvantages :-

i> It is time consuming process to access any arbitary data items or data element.
ii> It will consume more memory. If the number of fields are more, then more memory space is needed.

Q> WAP to insert a node after a given data value in linked list.

Ans

```
void addNodeAfterVal (int val) {
        temp = first;
    while (temp->data != val) { temp = temp->next; }
    ttemp = temp->next;
    P = (node*) malloc (size of (node));
    P->data = val;
    P->next = NULL;
    temp-> next = P;
```

```
    p->next = ttemp;
  }
```

Q> WAP to insert a node before a given data value.

Ans

```
void addNodeBeforeVal (int val) {
    temp = first;
    while (temp->data != val) {
        ttemp = temp;
        temp = temp->next;
    }
    p = (node *) malloc (sizeof (node));
    p->data = val;
    p->next = NULL;
    ttemp->next = p;
    p->next = temp;
}
```

Q> WAP to display the content of linked list.

Ans

```
void display () {
    temp = first;
    while (temp != NULL) {
        printf ("%d", temp->data);
        temp = temp->next;
    }
```

Q> WAP to delete the first node of linked list.

Ans

```
void deleteFirst () {  temp = first;
                       first = first->next;
                       temp->next = NULL;
                       free (temp); }
```

Q> WAP to delete last Node of linked list.

Ans
```
void deleteLast () { temp = first;
    while (temp -> next != NULL) {
        ttemp = temp;
        temp = temp -> next;
    }
    ttemp -> next = NULL;
    free (temp)
}
```

Q> WAP to delete node after a data value

Ans
```
void deleteAfterVal (int val) { temp = first;
    while (temp -> data != val) {
        temp = temp -> next
    }
    ttemp = temp -> next;
    P = ttemp -> next;
    temp -> next = P;
    ttemp -> next = NULL;
    free (ttemp);
}
```

Q> WAP to reverse a linked list.

Ans
```
void reverse () {
    P = first;
    Q = P -> next;
    R = Q -> next
    while (Q != NULL) {
        Q -> next = P;
        P = Q;
        Q = R;
        R = R -> next; }
} first = P;
```

Q> WAP to calculate sum of data of Linked list.

Ans
```
void sum () {
    temp = first;
    int sum = 0;
    while (temp != NULL)
    { sum = sum + temp -> data;
      temp = temp -> next;
    }
}
```

Doubly Linked List

```
typedef struct DoublyLinkedList {
        int data;
     struct DoublyLinkedList *prev, * next;
   } node;

node *first, *P, *temp, *ttemp ---;
  int count=0;
void createFirst (int val) {

     first = (node *) malloc (sizeOf (node));
     first -> data = val;
     first -> prev = NULL;     both Null allowed
     first -> next = null;
}

void addNode (int val) {
          temp := first;
     while (temp -> next != null) {
          temp = temp -> next;
     }
  ttemp = (node *) malloc (sizeof(node));
  ttemp -> data = val;
  ttemp -> prev = null;
  ttemp -> next = null;
  temp -> next = ttemp;
  ttemp -> prev = first;    count ++;
}
```

// inserting a node at middle of linked list

```
void addNodeAtMiddle (int val) {
        int c = 0;
        temp = first;
    while (c <(count/2)) {

            temp = temp -> next;
            c++;
        }
      P = getNode ( );
      P -> data = val;
      P -> next -> prev = NULL;
      ttemp = temp -> prev;
      P -> next = temp;
      P -> prev = ttemp;
      ttemp -> next = P;
      temp -> prev = P;
}
```

// delete a node before a given value

```
void deleteNodeBeforeValue (int val) {
    temp = first;
  while (temp -> data != val) {
        temp = temp -> next;
     }
   P = ~~temp~~ temp -> prev;
   ttemp = temp -> prev;
   //P -> next = ttemp; ~~ttemp~~
   ttemp -> next = temp;
   temp -> prev = ttemp;
   P -> next -> prev = NULL;
    free (P); }
```

P =
tte
te
tten
P -> ne
fr

Singly Circular Linked List

```
typedef struct SinglyCircularLinkedList {
                    int data ;
                struct SinglyCircularLinkedList *next ;
        } node ;

    node *first, *temp, *ttemp, *P ;

void createFirstNode (int val) {
        first = getNode ();
        first -> data = val;
        first -> next = first;
    }

void addNode (int val) {
        temp = first;
    while (temp -> next != first) {
                temp = temp -> next;
            }
    P = getNode ();
    P -> data = val;
    P -> next = first;
    temp -> next = P;
  }
```

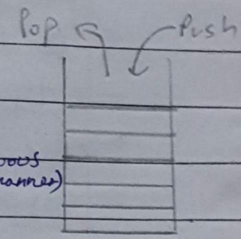
Doubly Circular Linked List

```
typedef struct DoublyCircularLinkedList {
        int data;
        struct DoublyCircularLinkedList *prev, *next;
    } node;

node *first, *P, *temp, *ttemp;

void createFirst (int val) {
        first = getNode ();
        first -> data = val;
        first -> next = first;
        first -> prev = ~~NULL~~ first;
    }

void addNode (int val) {
        temp = first;
        while (temp -> next != first) {
            temp = temp -> next;
        }
        P = getNode ();
        P -> data = val;
        P -> next = first;
        ~~ttemp = temp -> prev;~~
        P -> prev = temp;
        temp -> next = P;
        first -> prev = P;
    }
```

static → array → random accessing of elements
dynamic → linked list → sequential manner

Date ....
Page ....

Stack



* non primitive * linear * Abstract datatype *ordered list (Collection of elem in linear/continuous manner)
* Two operations :-
  i> Push (insert) → check maxsize → full (overflow)
  ii> Pop (delete) → empty → underflow
* LIFO principle → Last In First Out OR FILO → First In Last Out

```
// push value in stack
    int stack[maxsize]; top = -1;

    void push(int val) {
        if (top == maxsize-1) { printf("Stack Overflow!"); }
    else {
        top ++;   stack[top] = val;
        }
    }
```

```
// pop value from stack
        int pop(int val) {
            if (top == -1) { printf("Stack Underflow!");
                                                        }
            else { int poppedElement = stack[top];
                top --;
                return poppedElement; }
```

Operations performed on a Stack :-
i> Push () → insertion
ii> Pop () → deletion
iii> Peek () or top ()
iv> isFull ()
v> isEmpty ()
vi> Traversing ()
<vii> Sort()
<viii> Search()

Eg → messages, undo-redo, curly-braces checking in IDE.

Stack

* An array is a continuous block of memory while a stack is a LIFO data structure which allows access only to top of data.
* The value can be added or deleted on either side of array but in stack insertion & deletion is possible frome one side of the stack.

* The position indicator will be fixed at the top.
* The data is inserted at the top & removed from top.
* If there is access data then it is known as 'overflow' condition.
* If there is no data then it is known as 'underflow' condition.
* ~~Stack in which~~ If data is inserted, that operation is called 'push'.
* If data is removed, it is called 'pop' operation.

* There are two types of representation in stack i.e through (i) array implementation and (ii) Linked list implementation.

* Hence, A stack can be defined as a container in which insertion & deletion can only be done from one end i.e top of the stack.

* Stack is an abstract datatype with a predefined capacity which means that it can store the element of a limited type.

Standard stack operations:-

1) push- When we insert an element is stack then that operation

is 'push' and if the stack is full already the 'overflow' condition occurs.

2> pop - When we delete an item from the stack then that operation is known as the pop operation.
If the stack is empty it means that no element exists in the stack then this state is known as 'Underflow'.

3> isFull - It determines whether the stack is full or not.

4> isEmpty - It determines whether the stack is empty or not.

5> peek - It returns the element at the given position.
or top

6> display - It prints all the elements present in the stack.

7> search - It is used to find out the particular element in the stack.

8> count - It returns the total number of elements present in the stack.

// algorithm / pseudocode / steps for push() operation.

* Before inserting an element in the stack, we have to check whether the stack is full or not.
* If we try to insert ~~the~~ an element in the stack and the stack is full then the 'overflow' condition should occur.
* When a new element is pushed in the stack, the value of the top gets incremented and the element will be placed at the new position of the top.
* The elements will be inserted until we reach the maximum size of the stack.

// procedure for pop() operation.

i) Before deleting an element from the stack, we check whether the stack is empty or not.
ii) If we try to ~~dat~~ delete an element from an empty stack then the 'Underflow' condition should occur.
iii) ~~We will~~ If the stack is not empty, we will first access the element which is pointed by the top.
iv) Once the ~~pop~~ pop() operation is performed then top is decremented by 1.

Applications of stack:-
examples:-
* Deck of cards, piling of CDs / plates / books etc.
* Undo and redo operation.
* Recurssion (eg- Tower of Hanoi).
* Web histories, messages, phone book histories etc
* Polish notation (Infix, Prefix, Postfix)
~~Polish Notations~~ :-
* String reversal
* Tree traversal & DFS (Depth-First Search)
* Checking of the symbols in program (curly braces, square braces etc in IDEs)

// Implementation of stack using array
// static implementation of array.

```c
# define max 5;

typedef struct Stack {
        int data[max];
        int top;
    } S;

void init (S *ptr) {
        ptr -> top = -1;
    }

int isFull (S *ptr) {
        if (ptr -> top == max-1) return 1;
        else return 0;
    }

int isEmpty (S *ptr) {
        if (ptr -> top == -1) return 1;
        else return 0;
    }

void push (S *ptr, int val) {
        if (isFull (ptr)) printf ("Stack Overflow!");
    else {    ptr -> top++;
        ptr -> data[ptr -> top] = val;
    }
  }

int pop (S *ptr) { if (isEmpty (ptr)) printf ("Stack Underflow!");
    else { int val = ptr -> data [ptr -> top];
        top--;
        return val; }
    }
```

Polish Notation

Operators → (,) , + , − , * , ÷ , **
operands → A , B , Z or any no. 1 2 3 etc

Infix → A + B
Prefix → +AB
Postfix → AB+

Operators having lower priority can insert higher priority on Stack.
Operators having same priority one will get ~~get~~ popped which is on top (right) of stack & then new operator is inserted in Stack.

Precedence Rule
① Brackets
② Exponential (**)
③ Multiplication (*) / Division (/)
④ Addition (+) or subtraction (−)

Q) Convert the following infix expression to post fix expression.
① A + B * C / E − F

Ans

Table Method

Input	~~Stack~~ Stack	Expression
A		A
+	+	A
B	+	A B
*	+ *	A B
C	+ *	A B C
/	+ /	A B C *
E	+ /	A B C * E
−	−	A B C * E / +
F	Null	A B C * E / + F −

Ans → ABC * E / + F −

② (a + (b*c) /(d - e))

Ans

Input	Stack	Output (Expression)
(	(	
a	(	a
+	(+	a
(	(+ (	a
b	(+ (	a b
*	(+ (*	ab
c	(+ (*	abc
)	(+	abc *
/	(+/	abc *
(	(+/(	abc *
d	(+ /(	abc * d
-	(+ /(-	abc *d
e	(+/(-	abc*de
)	(+/	abc*de -
)	Null	abc*de -/+

Ans → abc*de -/+

or (a + (b*c) /(d - e))

(a + bc* / de -)

a + bc*de -/

abc*de-/+

3> (A /(B - C) * D - E)

Ans (A /(B - C) * D - E)

(A / BC- * D - E)

ABC-/ *D -E

ABC -/D* -E

ABC - /D * E -

Input	Stack	Expression
(	(	
A	(	A
/	(/	A
(	(/(	A
B	(/(	AB
-	(/(-	AB
C	(/(-	ABC
)	(/	ABC-
*	(*	ABC-/
D	(*	ABC-/D
-	(-	ABC-/D*
E	(-	ABC-/D*E
)	Null	~~ABC-/DE-*~~
		ABC-/D*E-

(4) A*(B*C+D*E)+F

Table method →

Input	Stack	O/P
A		A
*	*	A
(	*(	A
B	*(	AB
*	*(*	AB
C	*(*	ABC
+	*(+	ABC*
D	*(+	ABC*D
*	*(+*	ABC*D
E	*(+*	ABC*DE*+
)	*	ABC*DE*+
+	+	ABC*DE*+*
F	Null	$\boxed{ABC*DE*+*F+}$

Direct → method

A * (B * C + D * E) + F

A * (BC* + DE*) + F

A* BC* DE* + + F

ABC* DE* + * + F

$\boxed{ABC{*}DE{*}{+}{*}F{+}}$

⑤ (A + B) * C + (D − E) / F + G

AB+ * C + DE− / F + G

AB+C* + DE−F/ + G

AB+C* DE−F/+ + G

$\boxed{AB{+}C{*}DE{-}F{/}{+}G{+}}$

Input	Stack	O/P
(	(	
A	(	A
+	(+	A
B	(+	AB
)	Null	AB+
*	*	AB+
C	*	AB+C
+	+	AB+C*
(	+(	AB+C*
D	+(	AB+C*D
−	+(−	AB+C*D
E	+(−	AB+C*DE
)	+	AB+C*DE−
/	+/	AB+C*DE−
F	+/	AB+C*DE−F
+	+	AB+C*DE−F/+
G	Null	$\boxed{AB{+}C{*}DE{-}F{/}{+}G{+}}$

Date
Page

22 Sept 2022

Q> $(A+B)*C+D/(E+F*G)-H$

$AB+ * C + D/(E+FG*) - H$

$AB+C* + D/EFG*+ - H$

$AB+C* + DEFG*+ / - H$

$AB+C*DEFG*+ /+ - H$

$\boxed{AB+C*DEFG*+ /+ H-}$

For infix to prefix :- Applying Reverse Polish Notation

$\Rightarrow H - (G*F+E)/D + C*(B+A)$

$H - (GF* + E)/D + C* BA+$

$H - GF*E+ /D + C BA +*$

$H - GF*E+D/ + CBA+*$

$HGF*E+D/ - + CBA+*$

$HGF*E+D/- CBA+*+$

Now reversing again, $\boxed{+*+ABC-/D+E*FGH}$

~~$H - (*GF + E)/D + C* +BA$~~

~~$H - +*GFE / D + *C+BA$~~

~~$H - /+*GFED + *C+BA$~~

~~$-H/+*GFED + *C+BA$~~

~~$+-H/+*GFED * C+BA$~~

Q) a + b * c + (d * e + f) * g

infix to prefix → a + *bc + (*de + f) * g

+a*bc + +*def * g

+a*bc + *+*defg

$\boxed{++a*bc*+*defg}$

infix to postfix → a + bc* + (de* + f) * g

abc*+ + de*f+ * g

abc*+ + de*f+g*

$\boxed{abc*+de*f+g*+}$

Implement stack using linked list. // dynamic stack

```
typedef struct Stack {
        int data;
        struct Stack *next;
} Node;

Node* top, *temp;
top = NULL;
void push (int val) {
    if (top == NULL){
        temp = (Node*) malloc (sizeof(Node));
        temp -> data = val;
        temp -> next = NULL;
        top = temp;
    }
    else {  temp


void push (int val) {
        temp = (Node*) malloc (sizeof(Node));
        temp -> data = val;
    if (top == NULL)   temp -> next = top;
                       top = temp;
    }

int pop () {  if (top == NULL) { // underflow
                printf ("Stack is empty!");
              }
              int val = top -> data;
    else { printf (" %d popped from stack!\n", val top -> data);
           temp = top -> next;
           top -> next = NULL; free (top);
           top = temp; return val; }
}
```

29/9/22

Tower of Hanoi using recursion IMP (Application of stack)

It is a mathematical puzzle where we have three rods and 'n' number of discs.
The objective of the puzzle is to move the entire stack to the another rod by satisfying the following two rules:-

i) Only one disc can be moved at a time.
ii) Each move consists of taking the upper disc from one of the stack and placing it on the top of other disc/ another stack.
iii) No disc can be placed on the top of the smaller disc

H (n, A, B, C) → H (n−1, A, C, B)
H (n, A, B, C) → H (1, A, B, C)
H (n, A, B, C) → H (n−1, B, A, C)

n → no. of discs
A → source
B → intermediate
C → destination

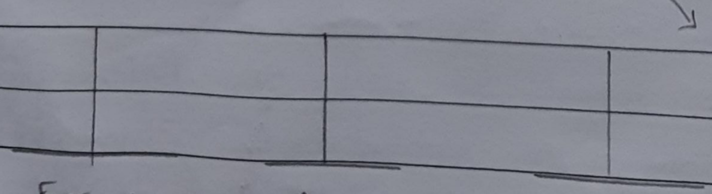
no. of steps = $2^n - 1$

H (3, A, B, C) → H (2, A, C, B)
- H (1, A, B, C) → A → C
- H (1, A, C, B) → A → B
- H (1, C, A, B) → C → B

H (3, A, B, C) → H (1, A, B, C) → A → C

A → C

H (3, A, B, C) → H (2, B, A C)
- H (1, B, C, A) → B → A
- H (1, B, A, C) → B → C
- H (1, A, B, C) → A → C



no. of disk → int → %d

From Aux To → char → %c

program :-

```
void Hanoi (int n, char From, char To, char Aux)
{   if (n = = 1) {
        printf ("Move disk %d from %c to
        %c to %c", n, From, To);
    }
else {
    Hanoi(n-1, From, To, Aux)
    printf ("Move disk %d from %c to %c", n, from to;
    Hanoi (n-1, Aux, To, From);
  }
}
```

Time complexity → $O(2^n)$ → cuz $2^n - 1$ calculations variable

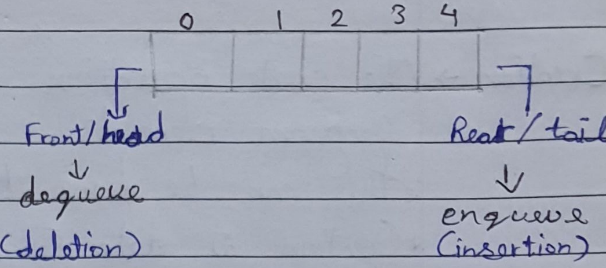
Space complexity → $O(n)$ cus no. of disks variable

other applications of stack → factorial, fibonachi

ADT ~~have~~ use less storage than other D's

Date ______
Page ______

Queue



DataStructure

* A datastructure is a way of organizing the data in the memory. It is the logical entity.
* We are having different data structures like stack, linked list, queues, arrays, trees and graphs.

Or we can say that, *

* It is the way of arranging data on a computer so that it can be accessed and updated efficiently.

* The choice of the datastructure is completely dependent on the programmer.

eg → If you want to store the data sequentially in memory, you will prefer the array datastructure.

Datatype

The datastructure that are predefined in system, are called primitive datatype. eg → int, float, bool, char etc.

* Datastructure are of two types:
  → Linear → Non-linear.

* Operations on Data Structure:-

i> Creation <iii> Deletion <v> Merging <vii> Searching
ii> Insertion <iv> Traversing <vi> Sorting

Creation → * The create operation results in reserving the memory for the program elements.
* This can be done by the declaration statement.
* Creation of data structure may either take place during Compile time or during runtime.
* For this we use, malloc(), calloc() and realloc() functions.

Destroy/ Delete → * Destroy operation destroys the memory space allocated for specified data structure.
* free() function is used to destroy it.

Updation/ Modification → It updates or modifies the data in data structure

Searching → It finds the presence of the desired data item in the list of data items. It may also find the location of all elements that may satisfy certain conditions

Sorting → It is the process of arranging all data items in a data structure in a particular order i.e either ascending or descending.

Merging → It is the process of combining the data items of two different sorted list into a single sorted list.

Traversal → It is the process of visiting each and every node in a list in a systematic manner.

Types of Linked List

Linked list basically is of four types :-

i) Singly Linked List
ii) Doubly Linked List.
iii) Singly Circular Linked List
iv) Doubly Circular Linked List

Singly Linked list → * It is a uni-directional linked list so you can traverse in only one direction i.e from head or start to tail or end.

* There are many applications for the singly linked list one common application is to store a list of elements that need to be processed in order.

* Singly linked lists are used in algorithms that need to process a list of items in reverse order.

Doubly Linked list → * It is a bi-directional linked list so we can traverse in both direction.

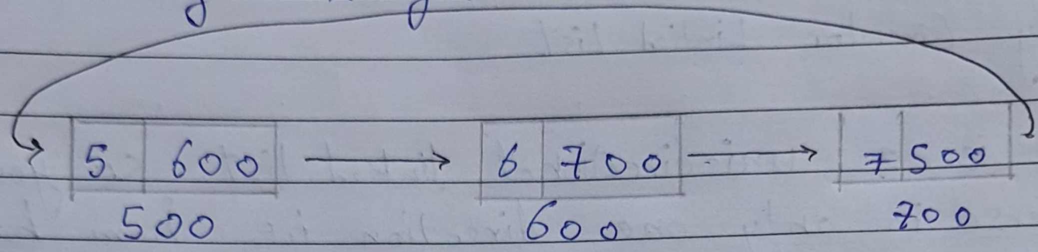
* Here the nodes contain one extra pointer called 'prev' pointer which points to previous node, its advantage is that it allows for quick insertion & deletion of elements.

* It is easy to implement and can be used in many application.

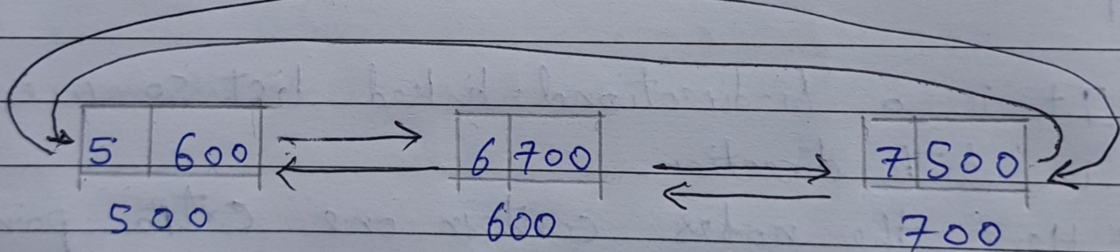
singly linked list ⇒ [500: 5 | 600] → [600: 6 | 700] → [700: 7 | NULL]

Doubly linked list ⇒ [500: N | 5 | 600] ⇄ [600: 500 | 6 | 700] ⇄ [700: 600 | 7 | N]

Circular linked list — It is a unidirectional linked list so you can only traverse in one direction but this type of linked list has its last node pointing to the first node so while traversing you need to be careful and stop traversing when you revisit the first node



Doubly Circular linked list — It is the mixture of doubly linked list & circular linked list.
Like doubly linked list, it has an extra pointer called 'prev' & similar to circular linked list its last node points to the first node, so this type of linked list is bidirectional in nature



Application of linked list:-

* It is used to represent
* It is used to navigate the websites
* It is used to implement stack & queue.
* We can implement an image viewer ~~with the~~ help ~~the~~ of Singly circular linked list.

compiler infix samajhte hai aur postfix me ~~ ~~ convert karke deta hai

Date
Page

Polish notation

In polish notation, the operator is placed before the operands, this notation is known as prefix notation.
But generally, we use the operator between two operands ~~then~~ it which is called infix notation.

Reverse polish notation.
In this the operator is placed before the operands

Types of notation :- / Stack traversal
There are three types of notation:
i) Infix
ii) Prefix
iii) Postfix

Conversion from infix to postfix expression :-

· There are some rules that we need to follow
i) If expression has parenthesis then the part inside it will be converted first.
ii) Operations will be converted in order of their precedence and associativity.

Conversion from infix to prefix expression :-

The rules will be similar to postfix but at the starting we need to reverse the expression and then solve it in the manner of postfix & whatever the ~~ ~~ result we get from it will again get reversed & ~~that~~ will be the prefix expression.

Q> a+b+c+d*e+f

Input	Stack	Expression
a	—	a
+	+	a
b	+	ab
+	+	ab+
c	+	ab+c
+	+	ab+c+
d	+	ab+c+d
*	+*	ab+c+d
e	+*	ab+c+d
+	+	ab+c+d*+
f	—	$\boxed{ab+c+de*+f+}$

Postfix expression is ab+c+de*+f+

Q> 5*3+4+1*2

15+4+2 = 21

Input	Stack	Expression
5	—	5
*	*	5
3	*	53
+	+	53*
4	+	~~54~~ 53*4
+	+	53*4+
1	+	53*4+1
*	+*	53*4+1
2	—	53*4+12*4

Q) p+q^r+s*t/(u-v)

Input	Stack	Expression
p	—	p
+	+	p
q	+	pq
^	+^	pq
r	+^	pqr
+	+	pqr^+
s	+	pqr^+s
*	+*	pqr^+s
t	+*	pqr^+st
/	+/	pqr^+st*
(	+/(	pqr^+st*
u	+/(	pqr^+st*u
-	+/(-	pqr^+st*u
v	+/(-	pqr^+st*uv
)	—	$\boxed{pqr\hat{}+st*uv-/+}$

p+q^r+s*t/(u-v)

p + qr^ + st* / uv-

p + qr^ + st*uv-/

pqr^++st*uv-/

$\boxed{pqr\hat{}+st*uv-/+}$

The postfix expression is pqr^+st*uv-/+

Q) 432+2^*5- Calculate its value

4 (5) 2^ * 5 -

4 (25) * 5 -

(100) 5 -

95

Abstract Datatype

* It is a mathematical modal.
* It is an abstraction of a datastructure that provides only the interface to which the data structure must satisfy.
* The interface does not give any specific details about Something should be ~~me~~ implemented or in which programming language ~~thou~~ it should get implemented.

* In other words ~~we~~ can say that the ADT are the entities that are the definition of data and operations but dox not have any implementation details.
In this case, we know ~~that~~ the data we are storing and the operations that can be performed on data but we do ~~k~~ not know its implementation details.

eg → A list is an abstract datatype that is implemented using array and linked list.
→ A queue and stack is an ADT and it is implemented using linked list based queue & array

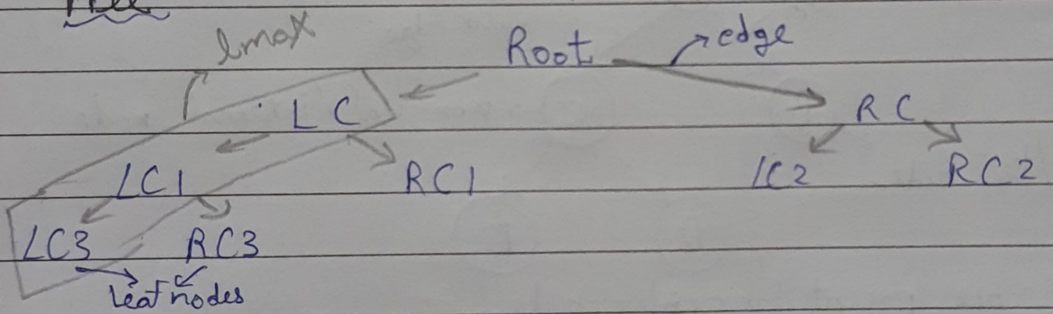
based queue.

Data structure

Linear
- → array
- → linked list
- → Stack
- → queue
- → single level
- → continuously placed

Non-linear
- → multilevel ; randomly placed
- → hierarical relationship
- → tree.

Tree



height of tree = lmax + 1 = 3 + 1 = 4

path to RC3 from root = 3 edges

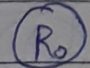
internal/Non leaf node = which divides

external/Leaf nodes = which doesn't divide

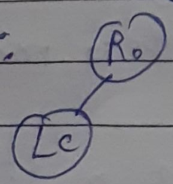
Root is neither considered Non-leaf nor leaf node.

Binary Tree

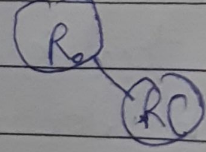
Case 1:



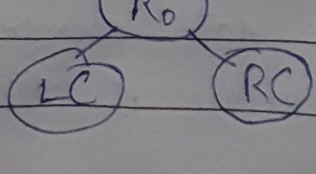
Case 2:



Case 3:



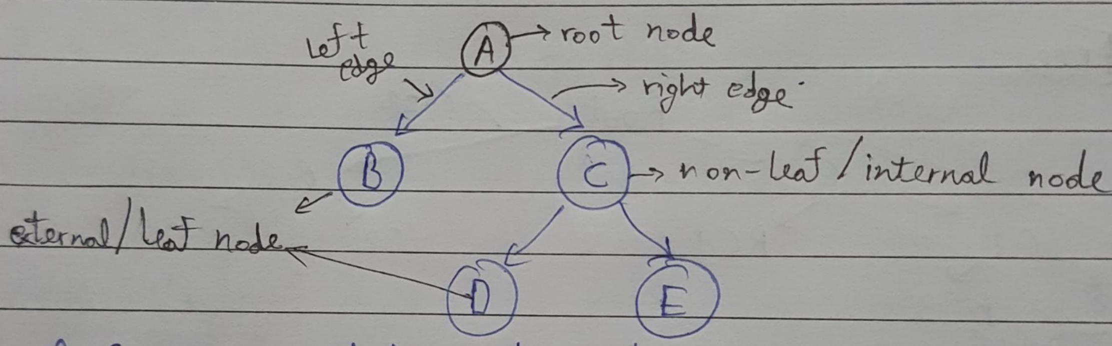
Case 4:



Tree

* A tree is a non-linear non-primitive data structure used to represent the parent-child relationship amongst its data elements. In other words, it is used to represent the heirarichal information.

* A tree is a collection of finite number of data element known as the nodes such that:
i> There exist a special node known as root node.
ii> All other remaining nodes are divided into disjoint subset such that each subset will again be a tree



A, C are parent / ancestor node
B, D, E are child / descendent node.

General Tree

A general tree datastructure has no limitations on the number of child nodes it can hold yet this is not the case with a binary tree.

Binary Tree

property / characteristic

* A binary tree is a datastructure with a maximum of two children for each parent.
* Every node in a binary tree has a left & right reference along with the data element.
* The node at top of heirarichy is called root node.

* A node that holds other sub-nodes are called ~~will be~~ ~~called~~ parent / internal / ancestor / non-Leaf node.
* A parent ~~as~~ has two ~~chide~~ child node, the left & right child so they can create a left sub-tree as well as right sub-tree.

root node ke right ko poora part.

Applications:-

~~eg~~ routers, drives in system, heap, hashing / mapping

i> hashing

ii> Routing in network traffic

iii> heap in memory.

iv> drives organization in the system.

Linked list representation of tree

Left child address	data	right child address

Node

Types of binary tree :-

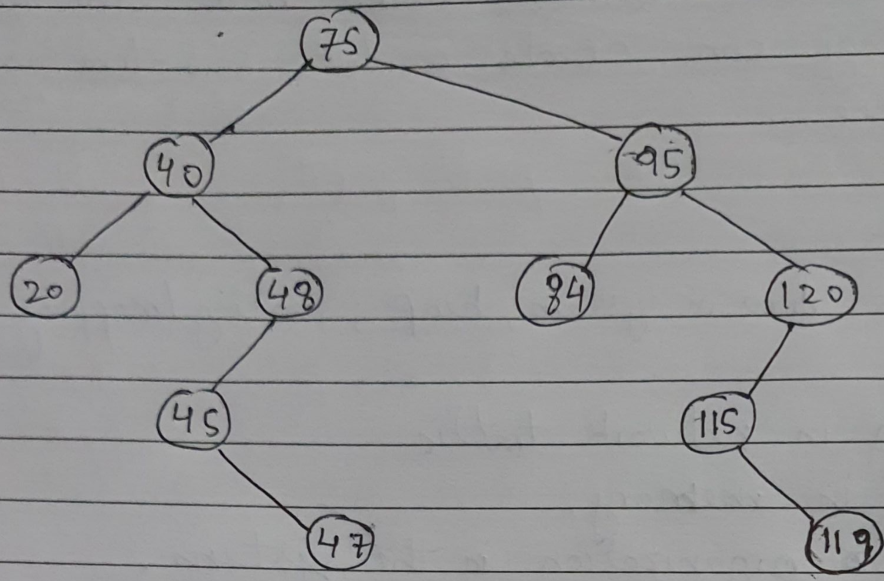
1> Full — 0 or 2 child
2> Perfect — Every level except root = 2 child
3> Complete — Perfect except last level can only have LC
4> Balanced — Left Sub tree - RST = 0
5> Degenrate — Only one child

Total no. of leaf nodes in a binary tree =

Traversal in binary tree:-

1> Inorder — LC R_o (Parent) RC
2> Pre order — R_o LC RC
3> Post order — LC RC R_o

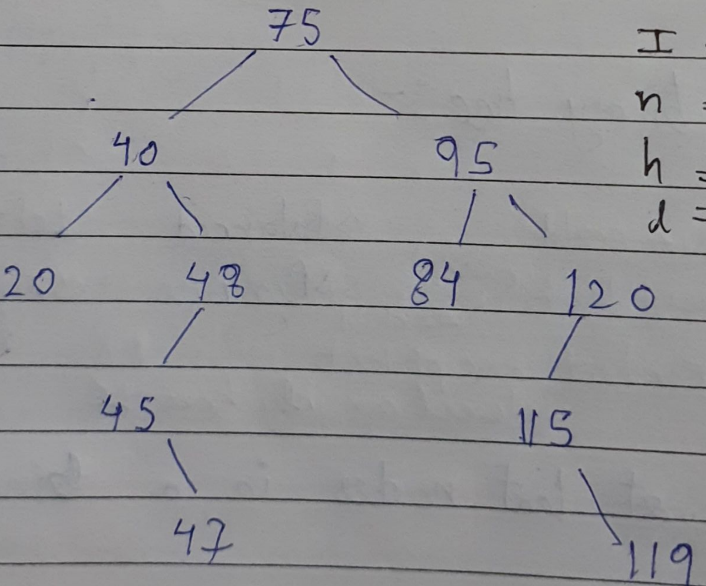
Q> Calculate inorder, preorder & post order for following binary tree:-



In order: 20 40 45 47 48 75 84 95 115 119 120

Pre order: 75 40 20 48 45 47 95 84 120 115 119

Post order: 20 47 45 48 40 84 119 115 120 95 75



I = internal nodes
n = total no. of nodes
h = height of tree
d = level of tree

In a full binary tree,

1) Given, Internal nodes (I)
leaf nodes = I + 1
Total nodes = n = 2(I) + 1

2> Given, total nodes (n)

Internal nodes, $I = \frac{(n-1)}{2}$

Leaf nodes =

3> Number of levels (l) is given

Total nodes, n =

Internal nodes, I =

Leaf nodes =

2> Given, total nodes n

Internal nodes, $I = \frac{(n-1)}{2}$

Leaf nodes =

3> Number of levels l is given

Total nodes, n =

Internal nodes, I =

Leaf nodes =

Algorithms :-

⇒ Algo Inorder (root) {

if (root == NULL) display "Tree is Empty";

Inorder root [Lchild];

display root [info];

Inorder root [Rchild];

}

⇒ Algo Preorder (root) {

Display root [info];

Preorder root [Lchild];

Preorder root [Rchild];

}

⇒ Algo Postorder (root) {

Postorder root [Lchild];

Postorder root [Rchild];

Display root [info];

}

Binary Tree

* It is a non-linear data structure with a maximum of two children for every parent.

* Every node in a binary tree has a left and right preference along with the data element.

* The node which is at the top of the heirarchy is called as the root node.

* The node that holds other subnodes is called parent node. (except root)
  A parent node has two child node, the left child and the right child.

Terms used in tree :-

Node - It represents a termination point in a tree.

Root - The tree's top most node is known as the root.

Parent/ Non-leaf/ Internal - Each node except root that has atleast one sub node is called a parent node / non-leaf node

Child/ External/ Leaf node - The nodes that have no child is called as the leaf node or a node that came from the parent node when moving away from the root is called as the child node.

Depth of tree - The number of edges from the tree's node to the root is called as the height / depth of the tree.

Components of tree

In binary tree, there are three components associated with it:

i) Data
ii) Pointer to left sub-tree
iii) Pointer to right sub-tree

Creation of structure using linked list.

```
typedef struct BinaryTree {
            int data;
      struct BinaryTree *left, *right;
} Tree;
```

Linked List representation of tree :- (Doubly)

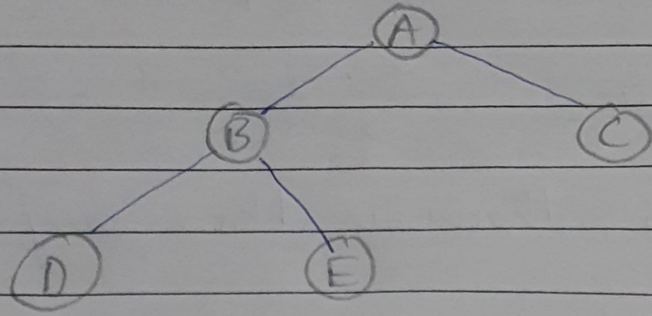
left	data	right

Types of Binary tree :-
There are various types of Binary tree & each has unique characteristics:

i) Full Binary tree

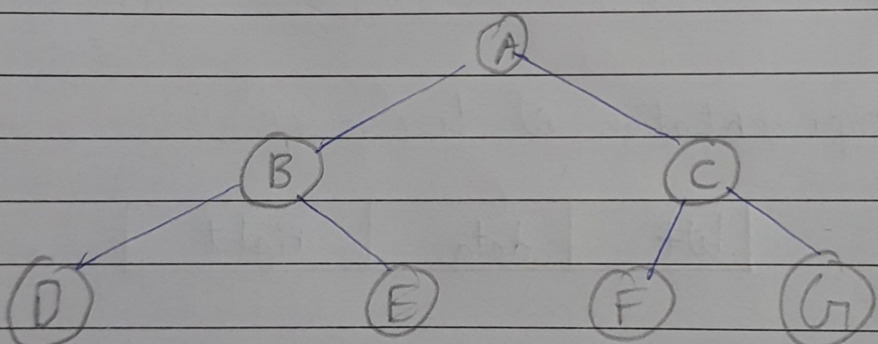
It is special type of binary tree that has either 0 or 2 children.
It means that all the nodes should either have two child node of its (parent) node or the parent itself

is leaf node or external node.



2) Perfect Binary Tree

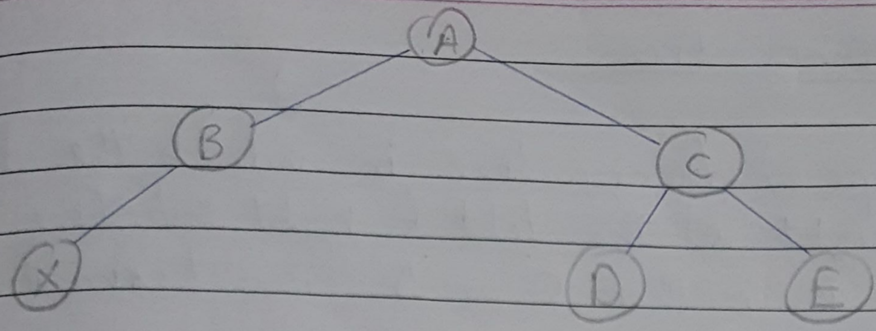
A perfect binary tree is a variant of binary tree & it is said to be perfect when all internal nodes have strictly two ~~chid~~ child nodes and every external node is at the same level or same depth in a tree



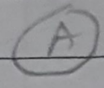
3) Complete Binary Tree

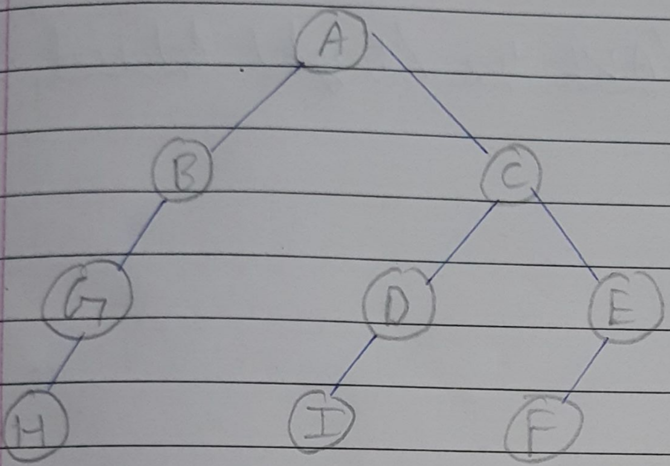
if node wants to split, it must've LC. AND All leaf nodes should be on same level

~~Each & every level except the last~~
A complete Binary tree is like a full binary tree but having two differences :-
i) Every level must be completely filled
ii) All the leaf nodes must bend towards the left.
i.e the last leaf element may not have the right sibling i.e a Complete binary tree does not have to be a full binary tree.

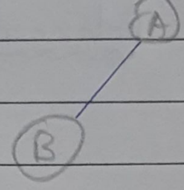


or





or

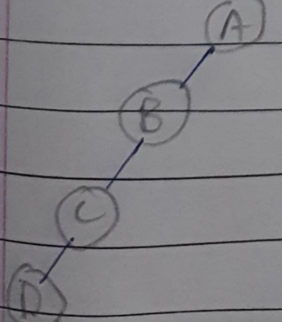


4> **Degenerate / Pathological / skewed binary tree**

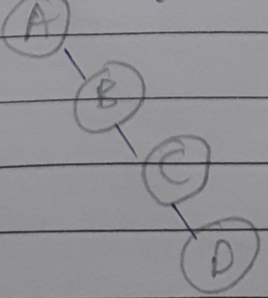
A ~~degerat~~ degenerate tree is a tree having a single child node, may be either left or right.

A skewed binary tree is a tree in which, the tree is either dominated by the left nodes or by the right nodes in binary tree so there are two types of skewed binary trees :- i> Left skewed tree

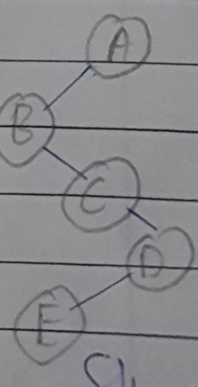
ii> Right skewed tree



Left skewed BT



Right skewed Binary tree



Skewed tree

5) Balanced binary tree

It is a type of binary tree in which the difference b/w the height of the left & right sub-tree for each node is either 0 or 1 is termed as balanced binary tree.

IMP

Balanced binary tree / AVL tree / height balanced tree

AVL tree can be defined as the height balanced tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right subtree from its left subtree i.e balanced factor = left child - right child

Tree is said to be balanced if the balance factor for each node is in between -1 to +1 i.e -1, 0, +1 then the tree is said to be balanced otherwise the tree will be unbalanced & need to be balanced, ideally the balance factor 0 indicates that the tree is perfectly balanced.

If the balance factor of any node is 1, it means that the left subtree is one level higher than the right subtree.

If the balance factor is -1, it means that the right subtree is 1 level higher than the left subtree.

If the balance factor is 0, it means that the left & the right subtree contain equal height.

Complexity of AVL tree:

The memory required will be equivalent to the total number of elements i.e $O(n)$.

The Searching time, insertion & deletion will take $O(\log n)$ time to execute.

Note → The time complexity will be completely dependent upon the height of the tree i.e $O(h)$.

Why AVL trees are required?

AVL tree controls the height of the binary search tree by not letting it to be skewed tree.

The time taken for all the operations in a binary search tree of height h is $O(h)$.

Operations on AVL tree:

It is also a binary search tree ∴ all the operations that are performed in the binary search tree will be same in AVL tree.

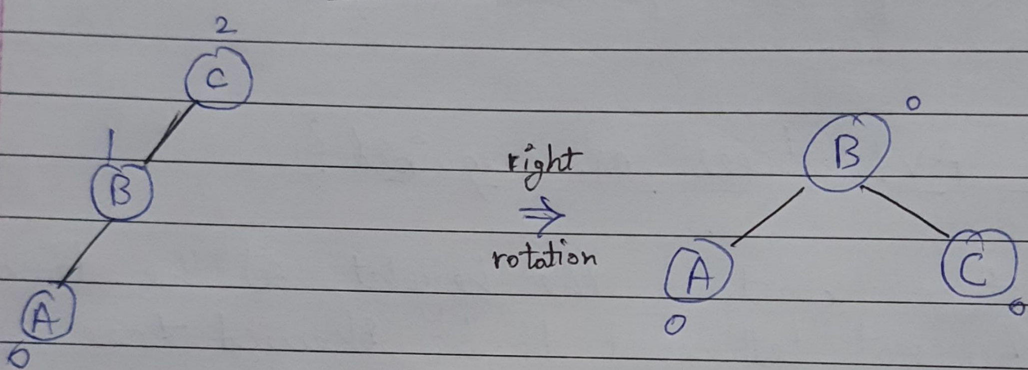
Searching & traversing do not lead to the ~~violatio.~~ violate the property of.
However, insertion & deletion operation can violate the property & therefore they need to be revisited.

AVL tree rotation

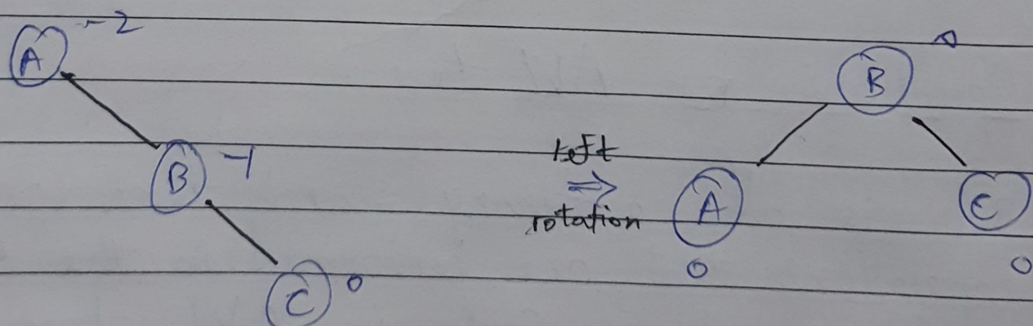
There are four types of rotations:-

i) Left - Left rotation (LL)
ii) Right - right rotation
iii) Left - right rotation
iv) Right - left rotation

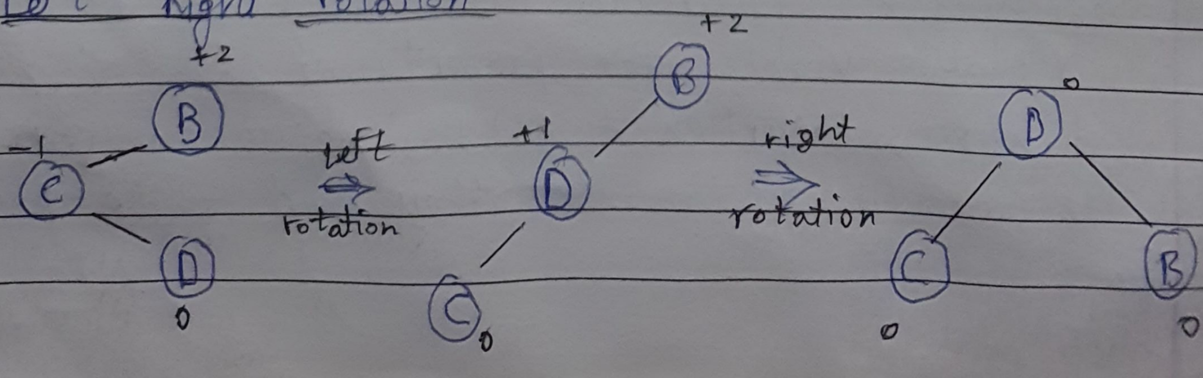
Left - Left rotation



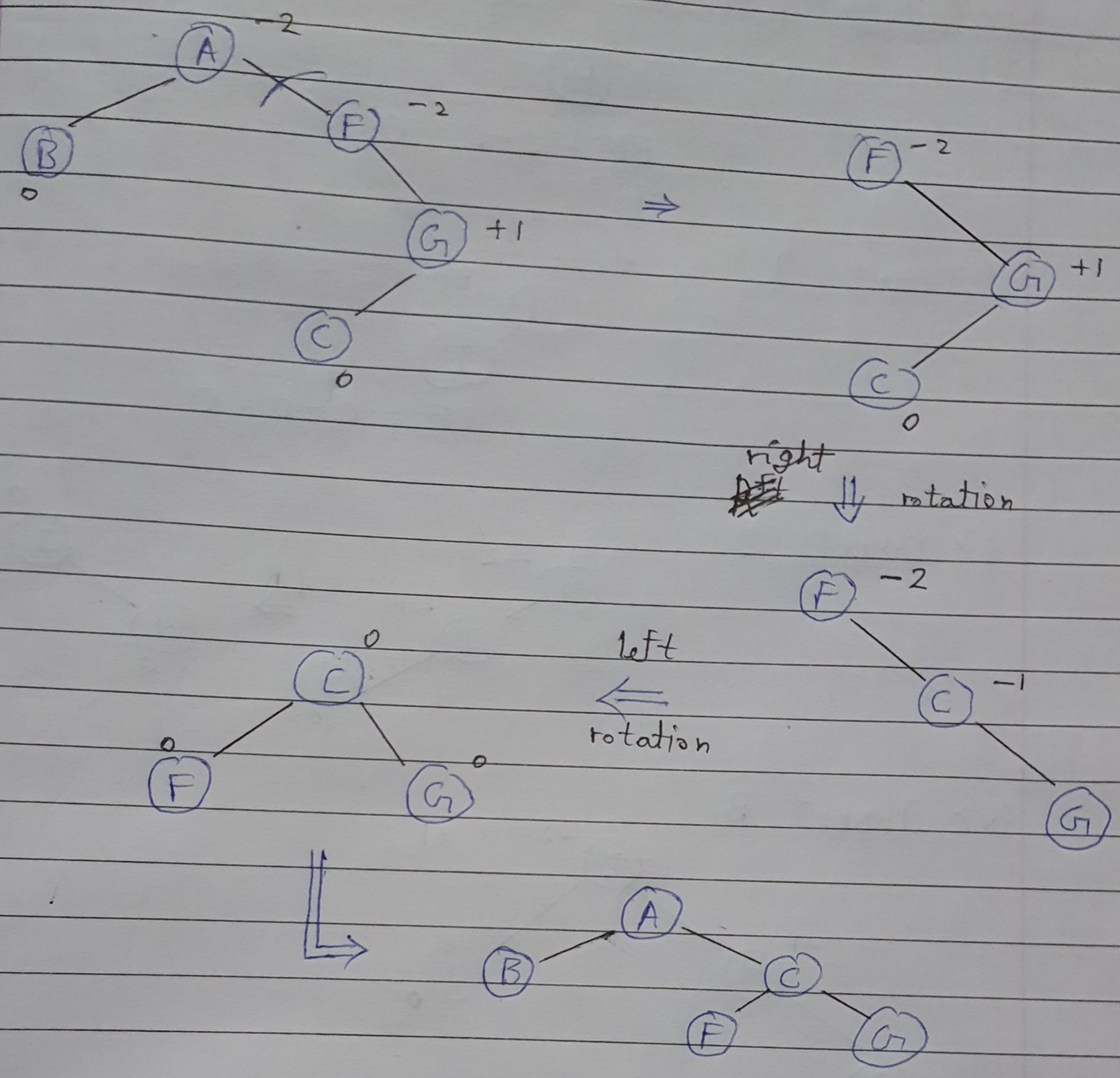
Right right rotation



Left - Right rotation



Right - Left rotation



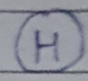
IMP Q) Construct an AVL tree for the following list of elements : H, I, J, B, A, E, C, F, D, G, K, L

Ans

Balance Factor = Left child − right child
Range = −1, 0, +1

For the left child, assign +1 and for the right child assign −1

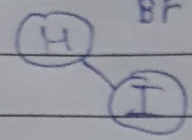
i) Insert H as a root node in tree



BF = 0-0 = 0

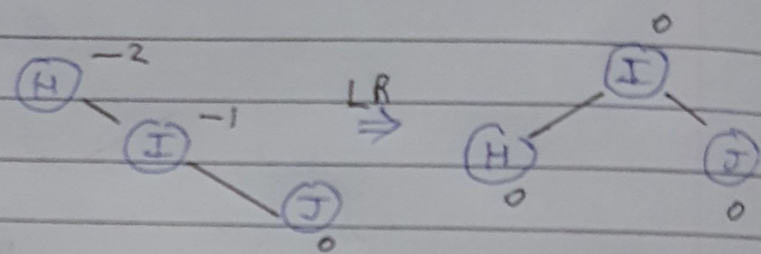
ii) Insert I in tree

BF = 0 -(-1) = -1

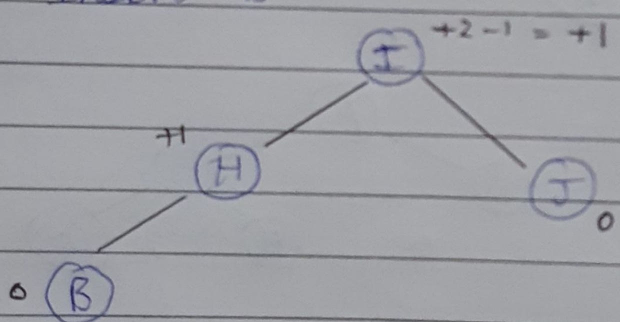


BF = 0-0 = 0

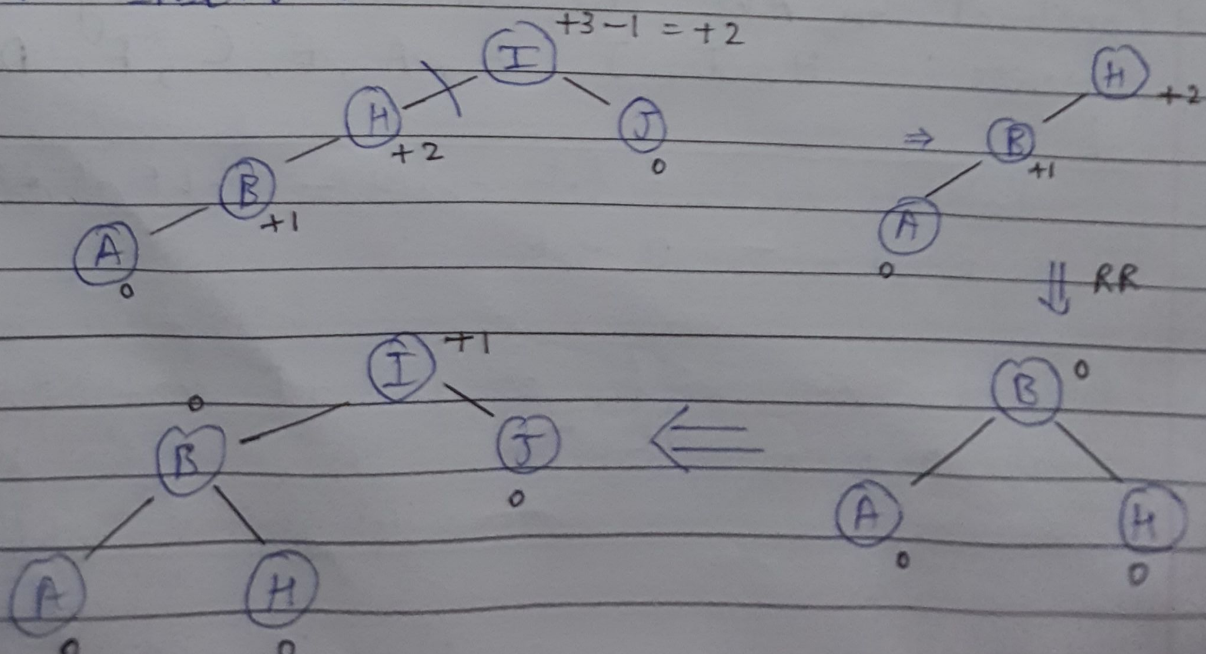
iii) Insert J



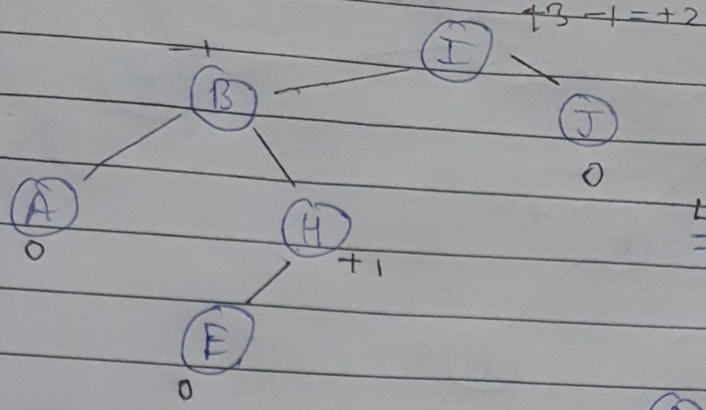
iv) Insert B



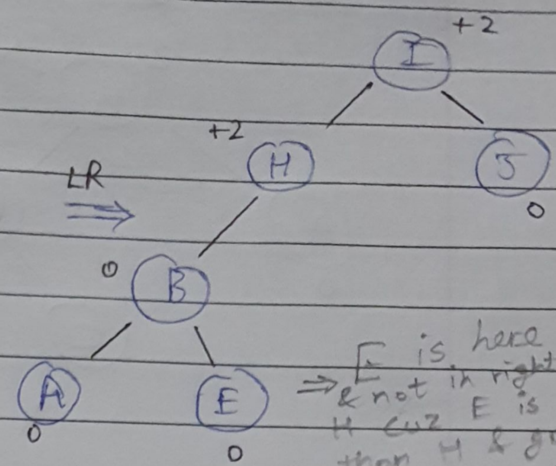
v) Insert A



vi) Insert E

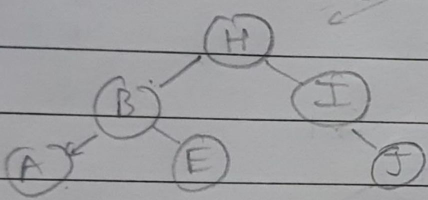


LR ⟹

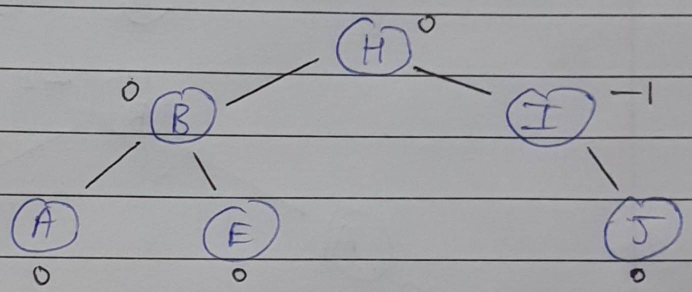


⟹ E is here & not in right of H cuz E is less than H & greater than B

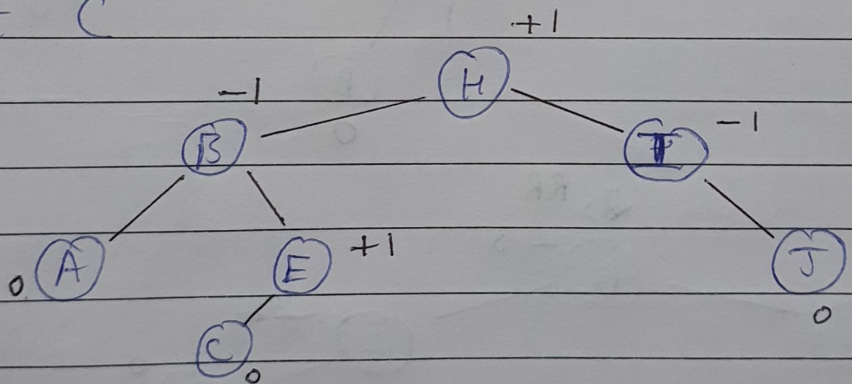
shortcut:-
arranging I, B, H is ascending order ⟹ B H I
⇓
making H root



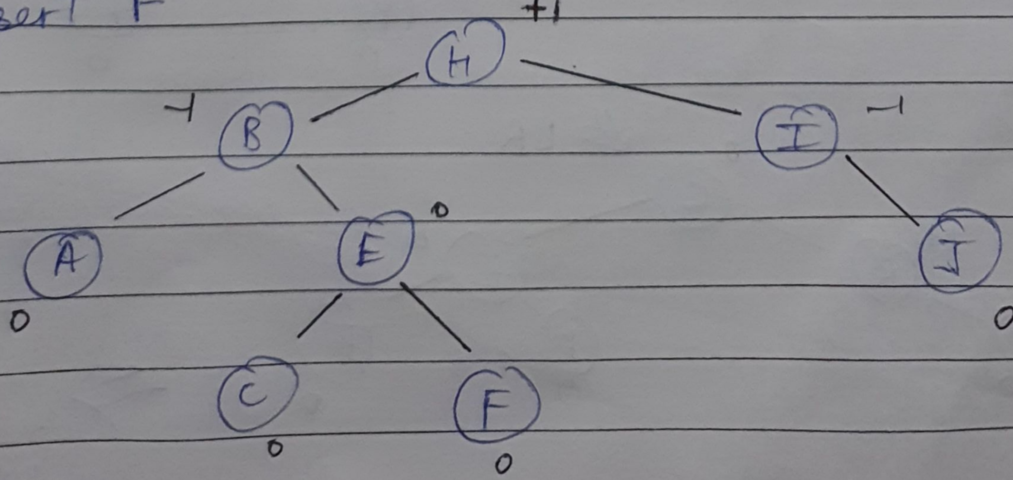
⇓ RR



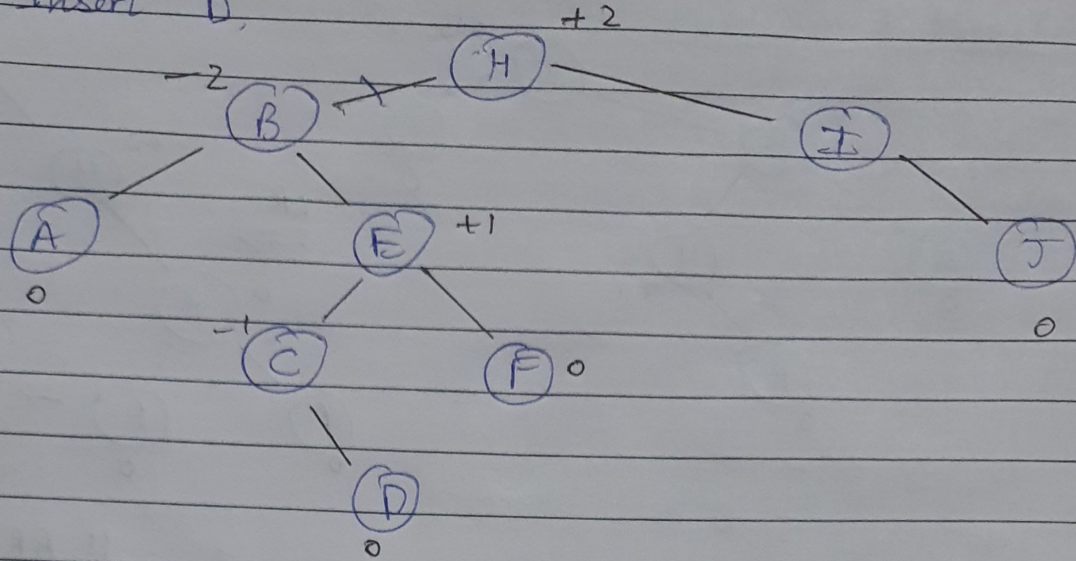
vii) Insert C



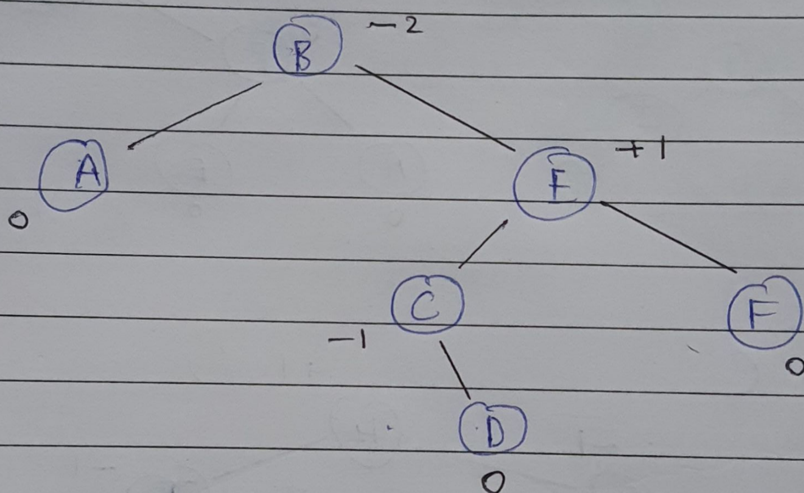
viii) Insert F



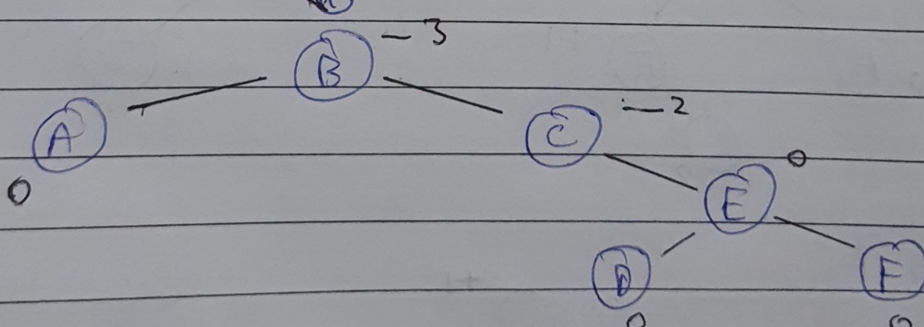
ix> Insert D.



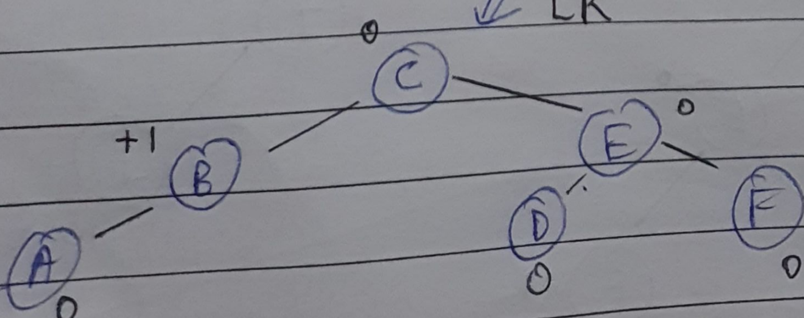
⇓



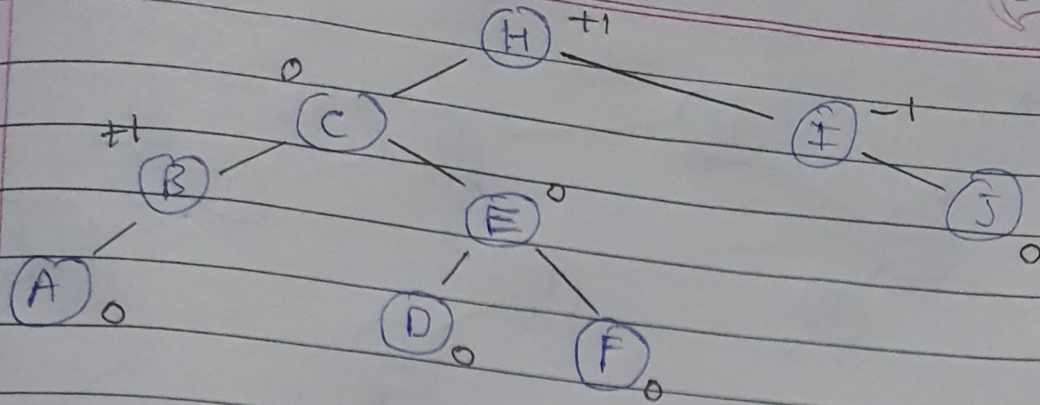
⇓ RR



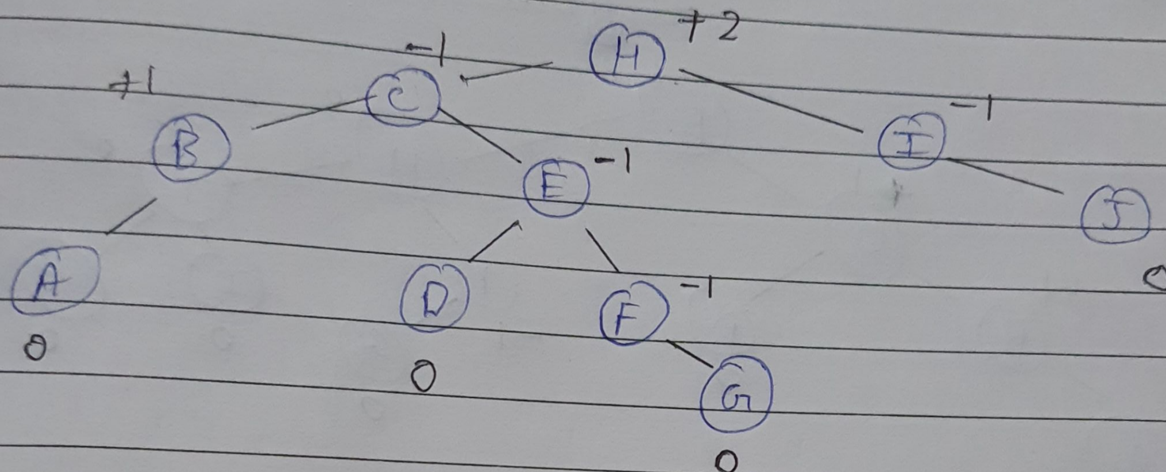
⇓ LR



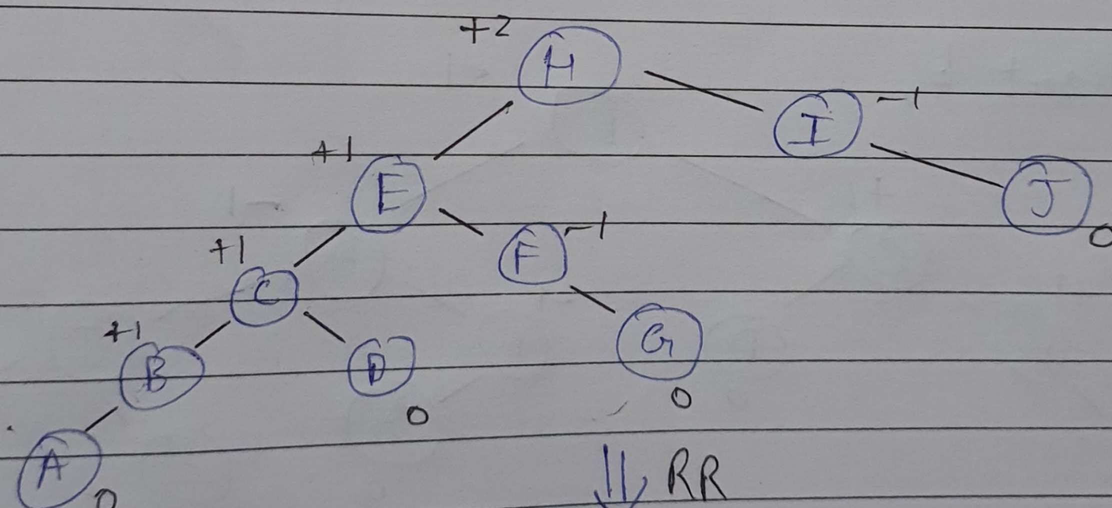
⇒



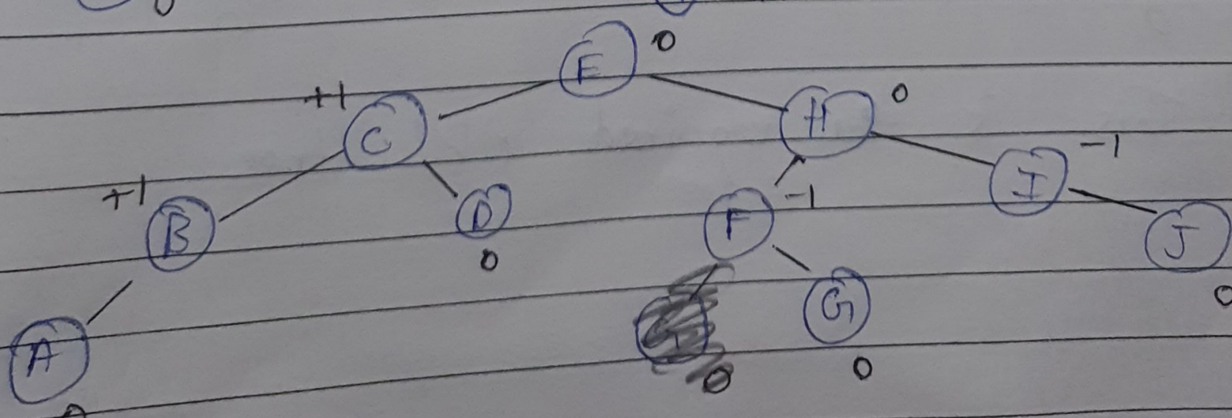
x> Insert G,



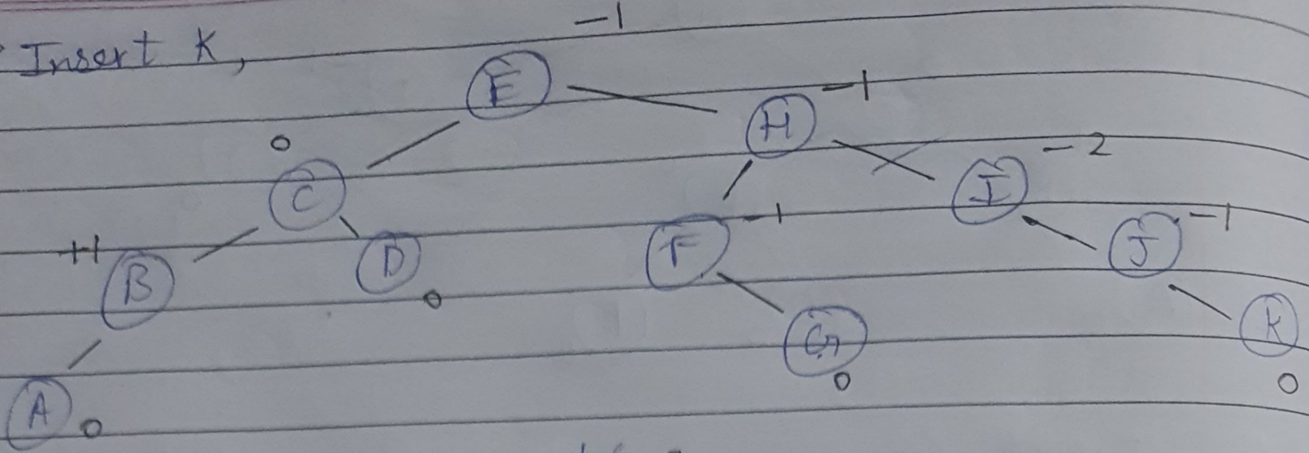
⇓ LR



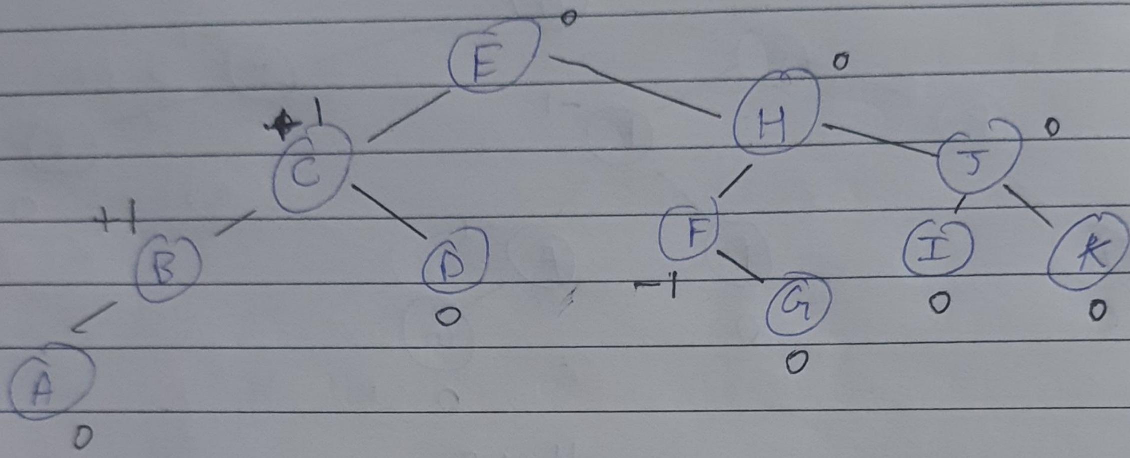
⇓ RR



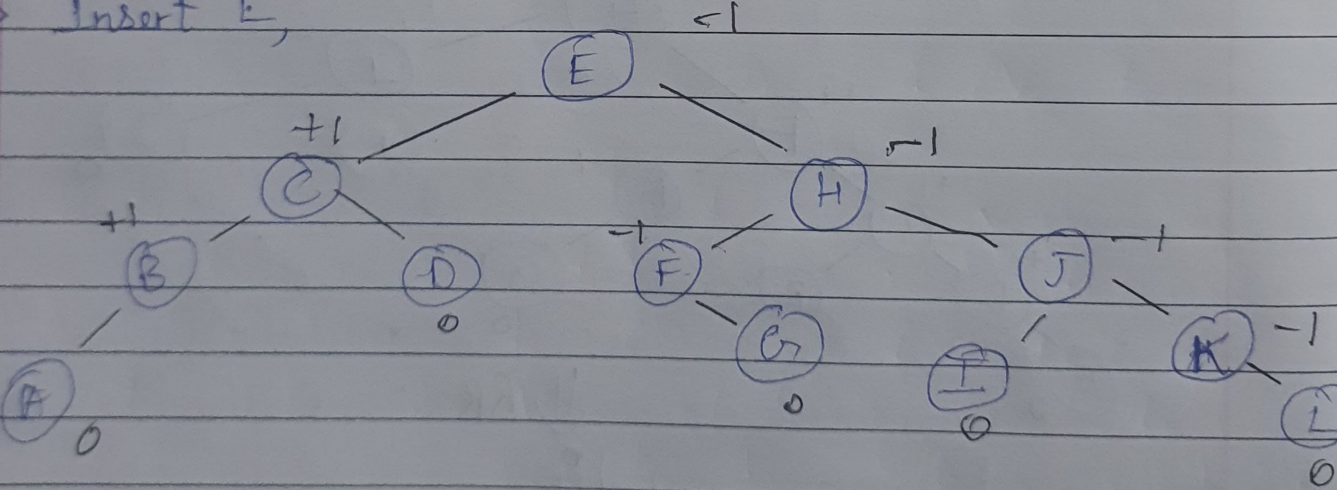
xi) Insert K,



⇓ LR



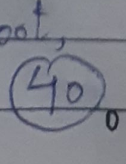
xii) Insert L,



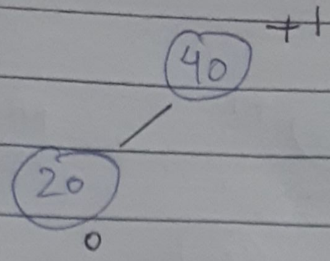
This is the required AVL tree

Q> Construct an AVL tree for the following list of elements: 40, 20, 10, 25, 30, 22

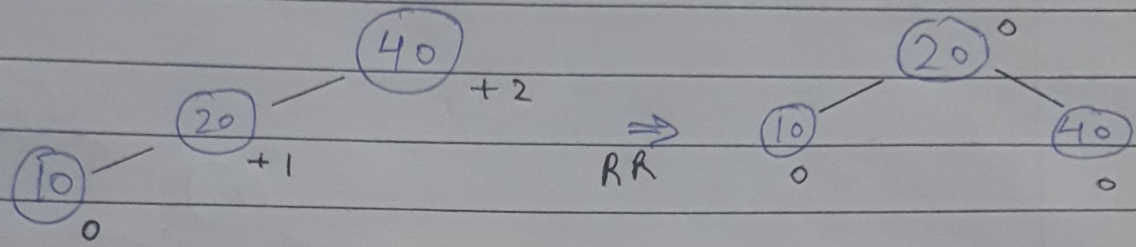
i) Insert 40 as root,



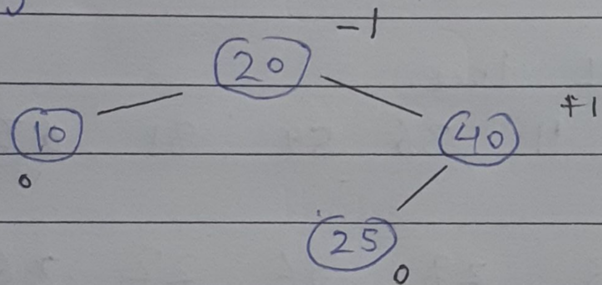
ii> Insert 20



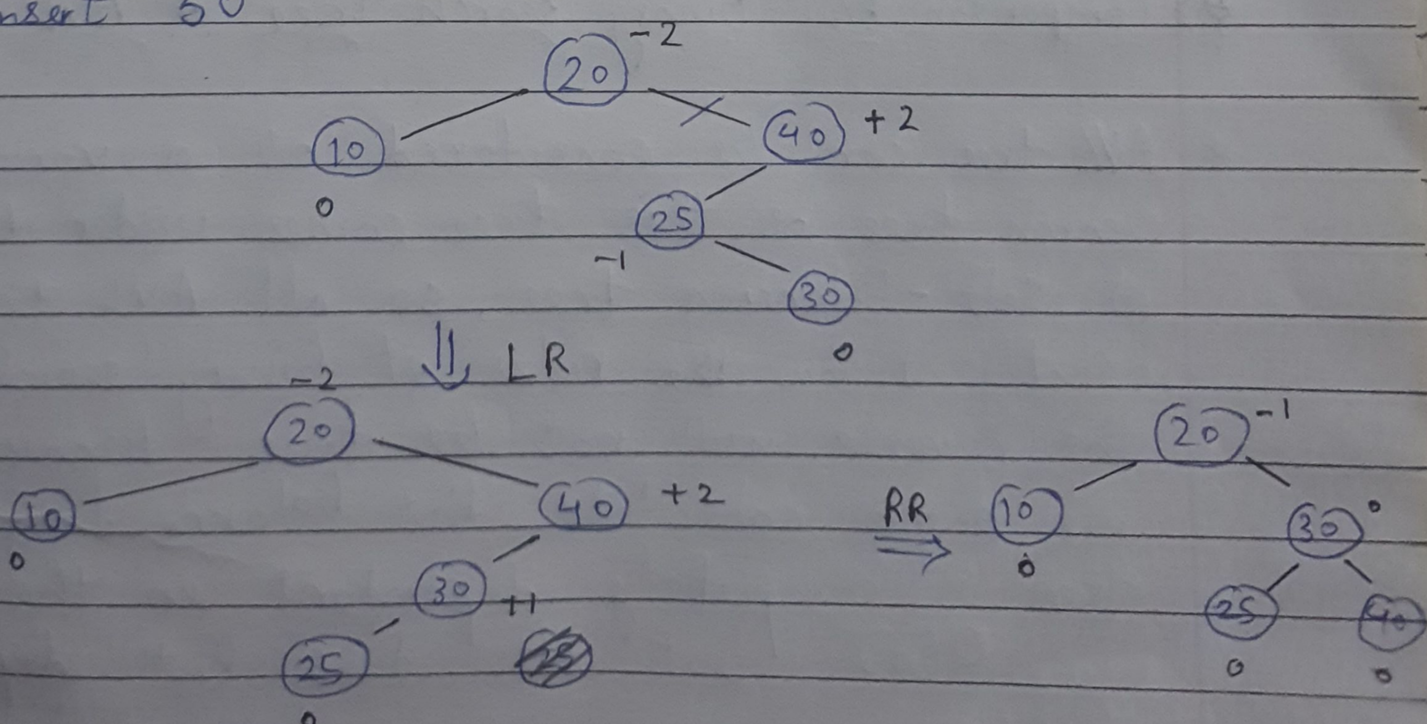
iii> Insert 10



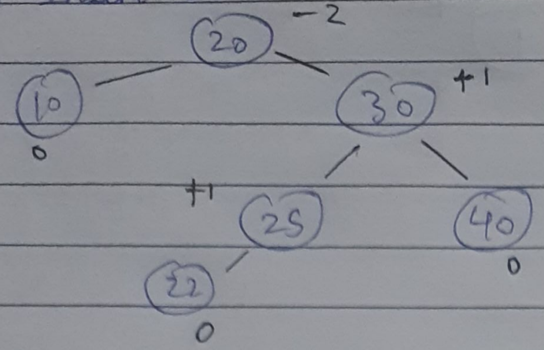
iv> Insert 25

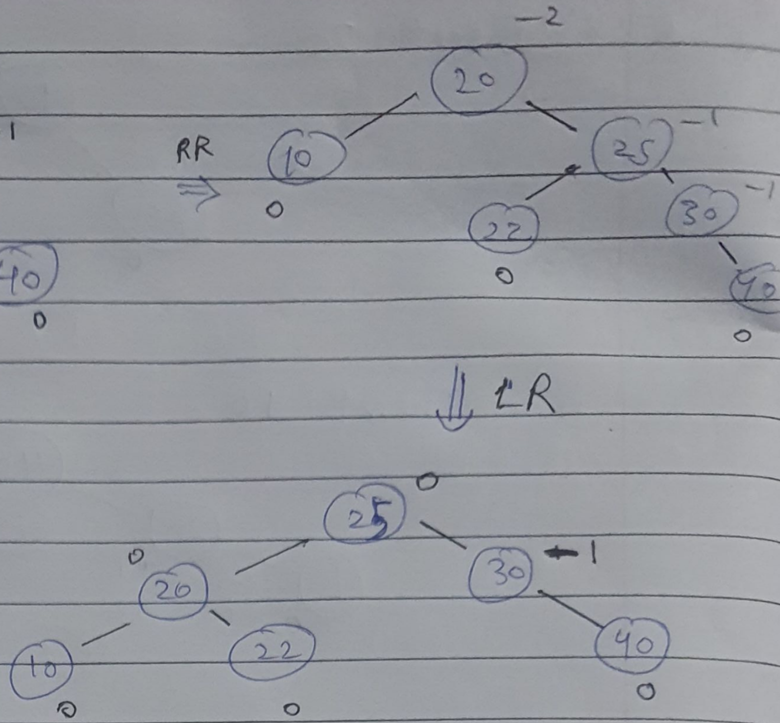


v> Insert 30



vi) Insert 22





Q) Construct AVL tree

16, 27, 9, 11, 36, 54, 81, 63, 72

Q) 30, 34, 23, 28, 36, 22, 26, 53

Comparison b/w binary search tree & AVL tree

* AVL tree can be considered as a variant of binary search tree as this tree comes under the category of self-balancing tree, so all AVL tree will exactly follow the rules of binary search tree but the reverse will not be true because for BST we don't calculate the balance factor & this tree is completely dependent on the no. of nodes i.e $O(n)$ but it does not depend on the height.

but In case of AVL tree it is guaranteed that for each & every level, the complexity will be exactly $O(\log n)$ for any operation (insertion, deletion, searching)

Disadvantages of AVL tree

* In case of AVL tree, If the no. of data elements are more then we have to apply more rotations in order to balance it, to overcome this problem, we make use of the concept of Red - Black tree.

*

* The AVL tree involves rotations & the concept of balance factor which makes it a complicated data structure

Advantages of AVL tree

* Its main advantage is its ability to self - balance itself. This self - balancing assures that the performance will be logn in worst case also.

* Finding the minimum & maximum element & the other operations that can be done using BST can also be ~~#~~ done with the help of AVL tree.

Applications of AVL tree,

* AVL trees are beneficial when designing some database application where the insertion & deletion operation is less than the searching operation.

* AVL trees are used for all sort memory allocation or collection in the form of sets.

Algorithm for insertion operation in AVL tree

0 & NULL is Same

```
Algo InsertBST (root, x) {   → for Algo
        iF (root == 0)
            {   root = new TreeNode();
                root[info] = x ;
                root[lchild] = NULL or 0;
                root[Rchild] = NULL or 0;
            }

        else if (root [info] > x) {
                Insert BST (root[lchild], x);
            }

        else { Insert BST (root [rchild], x);
            }
}
```

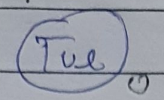
Algorithm for searching operation in AVL / BST

```
Algo SearchBST( root, x) {
            if (root == 0)   { PF ("Element not print")
    else if (x < root)
                { Search BST (root [LC], x)}

    else if (x > root) { Search BST (root[RC], x)}

else if (root[info]==x)
            { return PF ("Element Found!")
                  return 1 }

    return -1;
}
```

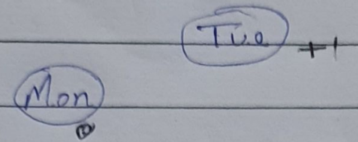
Q> Construct an AVL tree for the following:
Tue, Mon, Thur, Sat, Sun, Fri, Wed

Arrange Alphabetical ⇒
Fri
Mon
Sat
Sun
Thur
Tue
Wed

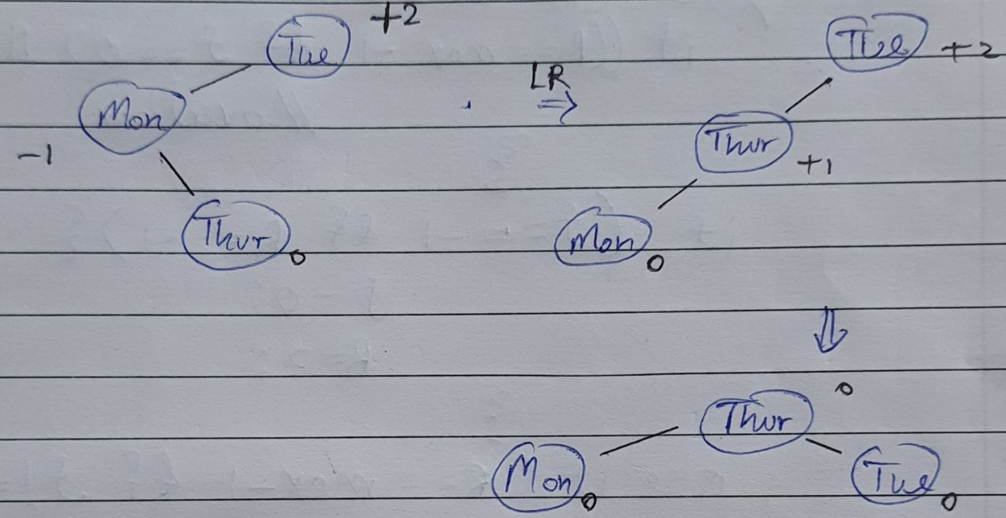
i>



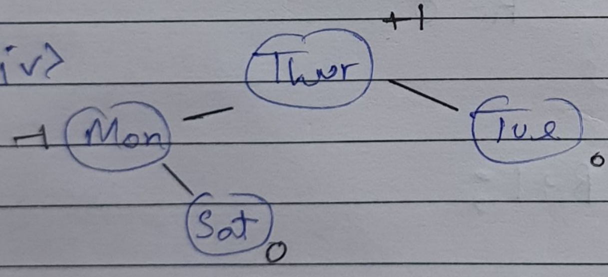
ii>



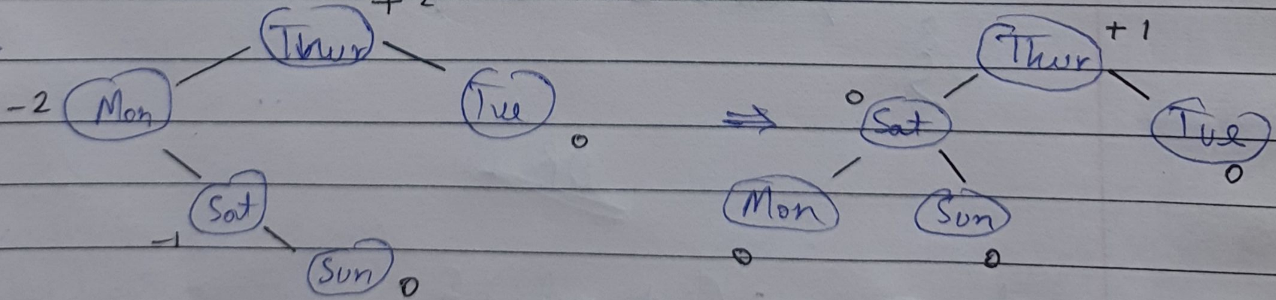
iii>



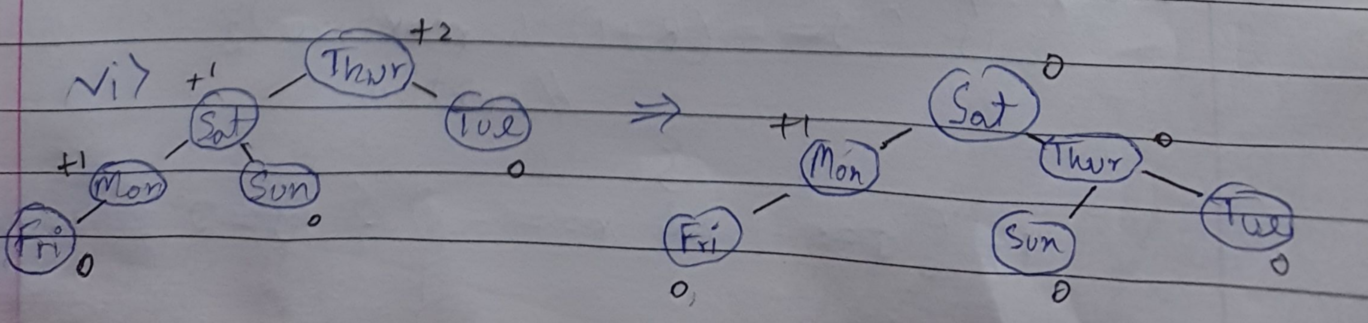
iv>



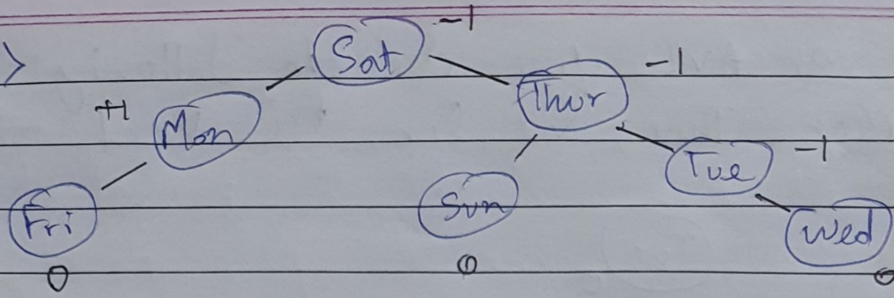
v>



vi>



vii>



Circular Queue

```
void enqueue (int val){
        if ((R==max -1 && f==0) || (r== f-1) ){
                    // overflow
            }
        if (f==-1 && R==-1){
                f=0;
                R=0;
            }
        else if (R==max-1 && f!=0){
                R=0;
            }
        else {
            R= R+1;
            }
        queue[R] = val;
}
```

OR

```
void enqueue (int val) {
    if ( f== (R+1) % maxSize ){
            // overflow
        }
    if ( f==-1 && r==-1) { f=0; r=0;}
```

else {R = (R+1) % maxSize;}
queue [R] = val;
}

Implementation of stack using Queue

top →	60	2
	40	1
	10	0

Q1 (Main)

60	40	10
0	1	2

Q2

40	10	
0	1	2

(auxilary)
F=R=-1

Insert → 10, 40, 60
Deletion → 60, 40, 10

Implementation of Queue using Stack

Required →

10	40	60
F ↗		↖ R

enqueue (10)
" (40)
" (60)

S1 → push(10)
S1 → push(40)
S1 → push(60)

S1	
60	← top
40	
10	

→

S2
10
40
60

S2 → push(S1 → pop()) // 60
S2 → push(S1 → pop()) // 40
S2 → push(S1 → pop()) // 10

dequeue () = S2 → pop() & then refill S1 & empty S2

Queue as a datastructure

Queue is a non-primitive linear data structure that is open ~~t~~ at both the ends and the operations are performed based on the FIFO principle (First In First Out)

We can define a queue to be a list in which all the additions ~~to~~ are made to one end that is called the rear & the deletion from the other end called the front.

* The queue can handle multiple data.
* We can access both the ends of it
* They are fast as well as flexible

Queue can be implemented in two ways, by using array & by using linked list.

To insert any element to the queue, we perform enqueue() operation & to delete any element we perform dequeue() operation.

Types of queue:

1> Linear queue
2> Double-ended queue
3> Circular-queue
4> Priority queue

heap	→ shrinks / contracts
stack	→ grows / expands
data	} fixed
text	

Date
Page

Heap

A heap is a tree based data structure & can be classified as a complete binary tree.

All the nodes of the heap are arranged in specific order.

In the heap data structure, the root node is compared with its children and arranged according to the order so if (A) is the root node & the (B) is its child then the property will be ~~key value~~ $\boxed{key(A) \geq key(B)}$ and will generate a Max Heap.

The above relation b/w root & child node is called as heap property.

Depending on the order of parent child node, the heap is generally of two types:-
i) Max heap
ii) Min heap

Max heap → In a Max Heap, the root node key is the greatest of all the remaining keys in the heap.

It should be ensured that the same property is ~~the~~ true for all the sub-trees in the heap.

Min heap → In this, the root node key value is the smallest among all the remaining keys present in the heap.

This property should be recursively true in all ~~the~~ the other sub-trees in the heap.

stack → local var → memory [illegible]

heap → global var → dynamic mem allocation

Applications of heap :

A heap data structure can be used in following areas :-

* To find out the k^{th} smallest or the k^{th} largest or smallest element in the heap.

* It is used to implement the priority queue.

* It is used in the embedded system like in the kernel to update the version of the operating system or it is used in unix or linux operating system to implement the security features.

Q> Construct heap for following list of elements
77, 15, 91, 21, 6, 46

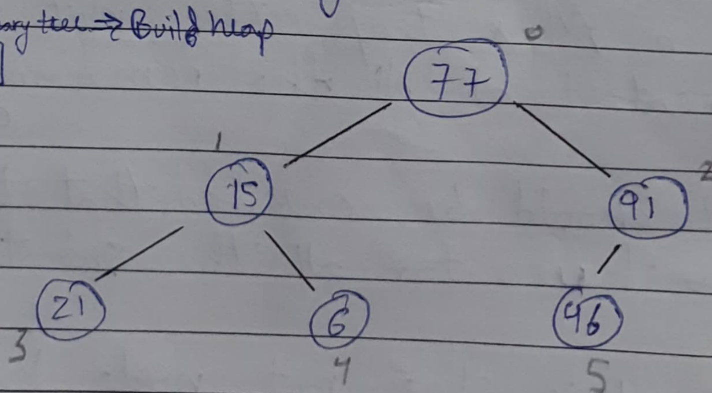
77	15	91	21	6	46
0	1	2	3	4	5

Array

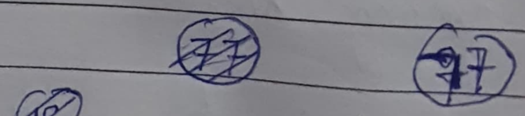
[make ~~heap~~ complete binary tree ⇒ Build heap]

$LC = 2n + 1$

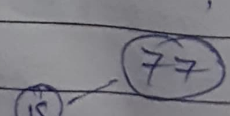
$RC = 2n + 2$



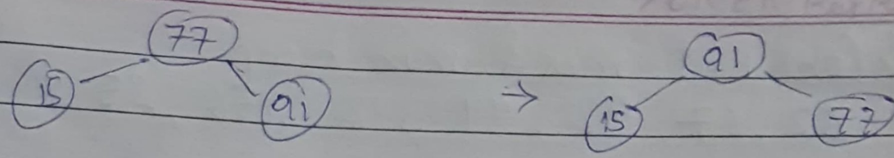
Build Max heap, i>



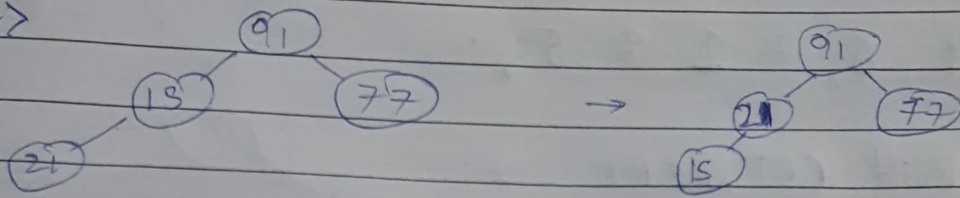
ii>



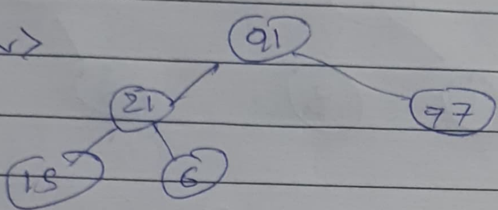
iii>



iv>

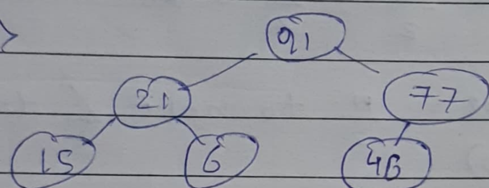


v>



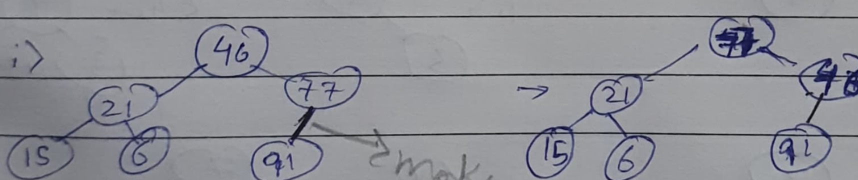
Heap Sort
↓
Unstable

vi>



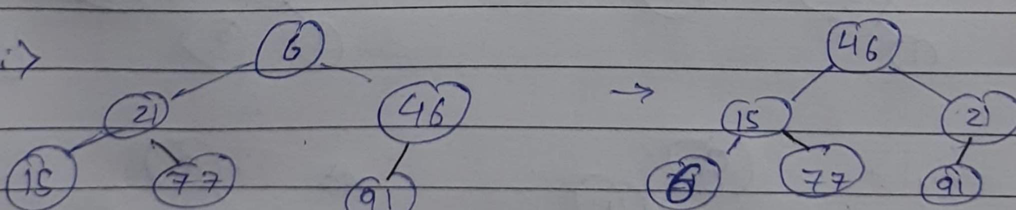
Build Min heap, (from max heap)
[~~Swa~~ After heapifying, swap the root & bottom-right-most method]

i>

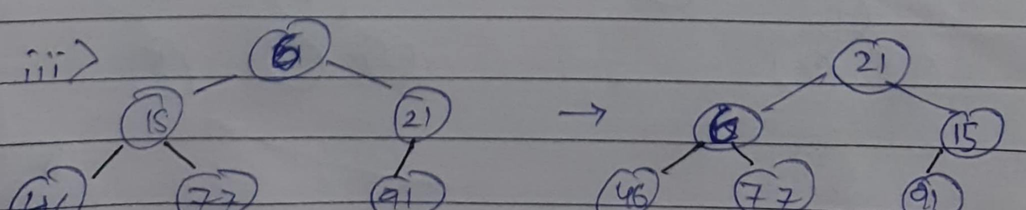


mark,
dotted → means deleted → stored in array from last

ii>



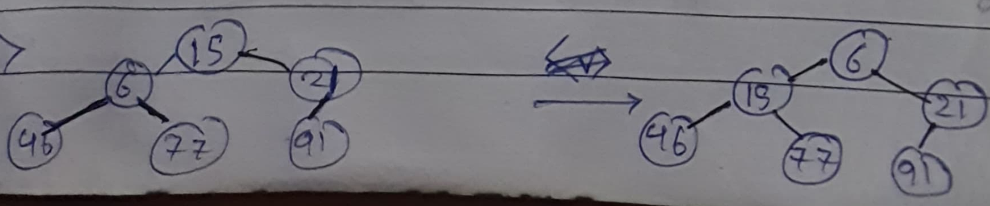
iii>



find array:

0	1	2	3	4	5
6	15	21	46	77	91

iv>



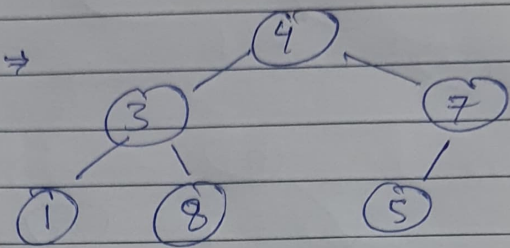
Heap → Two property → * Structure → CBT or ACBT
complete / Almost complete
* Ordering

Jitne levels utne maximum swapping

Q) Construct max & min heap
32, 67, 82, 44, 12, 56, 25

Q) 4, 3, 7, 1, 8, 5,

Build CBT/ACBT ⇒



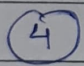
$n = 6$

Total leaf nodes $= \frac{n}{2} = \frac{6}{2} = 3$

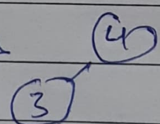
Range of leaf nodes $= \frac{n}{2}$ to $n-1 = \frac{6}{2}$ to $6-1 = 3$ to 5
(indexing from 0)

(indexing from 1) $= \left(\frac{n}{2}+1\right)$ to $n = 4$ to 6
Floor value

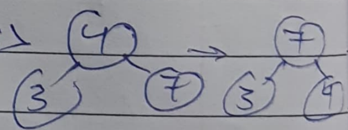
i)



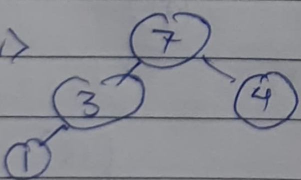
ii)



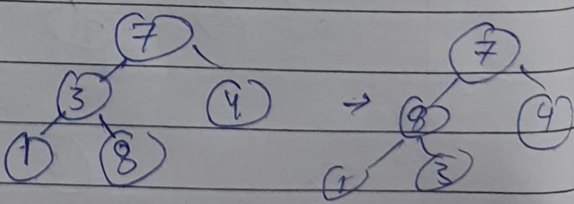
iii)



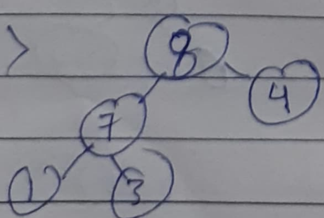
iv)



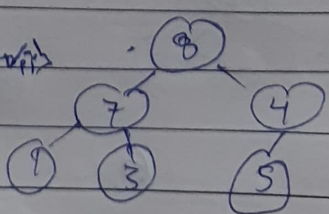
v)

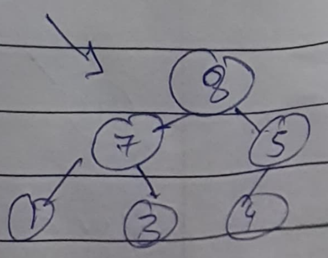


vi)



vii)





8 7 5 1 3 4 ⇐

```
Algo MaxHeapify (A, n, i) {

        int largest i;
        int l = 2i+1;
        int r = 2i+2;

    while (l ≤ n  && A[l] > A [largest]{
        largest = l;
    }
    while (r ≤ n  && A[r] > A [largest]) {
        largest = r;
    }

    if (i != largest {
            swap (A[i], A [largest]);
            MaxHeapify (A, n, largest);
        }
}

Algo HeapSort (A, n) {
        // Build Heap
    for (i = n/2 ; i ≥ 0 ; i--) {
            MaxHeapify (A, n, i);
        }

        // Deletion
    for (i = n - i ; i ≥ 0 ; i--) {
            swap (A[i], A[0]);
            Max Heapify (A, n, 0);
        }
}
```

Heap Sort

Heap Sort is divided into two parts :-

i) Creating a heap of the unsorted array.

ii) Then the unsorted array is repeated removing the largest & ~~smalling~~ ~~sot~~ smallest element from the heap & inserted it in array.

The heap is reconstructed after each removal.

* Initially on receiving ~~&~~ an unsorted array the first step in the heap sort ~~is~~ the creation of heap datastructure i.e either Max heap or min heap.
* Once the heap is built, the first element of the heap is either largest or smallest so we put the first element of the heap in our array at the last position.
* So we again make a heap using the remaining elements and ~~will~~ we again pick the first element of the heap & put it in the array.
* We will repeat the steps again until we will get complete sorted array.

The complexity of heap sort is $O(n \log n)$.

chunks → blocks sizes → from secondary to primary memory

block ptr, record ptr, key

B-Tree / Balanced Tree / Dynamic Multilevel indexed Tree

* It is a self-balancing tree data structure that keeps data sorted and allows searching, sequencial access, insertion & deletion in ~~logat~~ logarithmic time.

* The B-tree is a generalization of a binary search tree in that a node can have more than two children.

~~Unlt~~

* Unlike self balancing binary search tree, B-tree is optimized for the system that reads & writes large blocks of data.

* B-tree is a good ~~etc~~ example of data structure for external memory.

* It is commonly used in database & in file system. It is widdly used for dis-access

* Its main advantage is its capability to store large number of keys, values by keeping the height of the tree relatively small.

* A B-tree of order (n) contains all the properties of an multiway tree (M-way tree).

* The following are the properties of the B-tree:-

i) Every node contains atmost (m) children.
ii) Every node in the tree except the root node and the leaf node contain atleast (m/2) children

median element

leaf nodes at same lvl i.e left & right s...

insertion starts from leaf node bottom up

iii> The root-node must have atleast two nodes

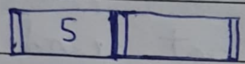
iv> All the leaf nodes must be at the same level.

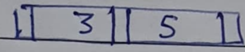
Children	root	intermediate / non-leaf node
max	m	m
min	2	$\lceil \frac{m}{2} \rceil$
min key-values →	1	$\lceil \frac{m}{2} \rceil - 1$
max key-values →	m−1	m−1

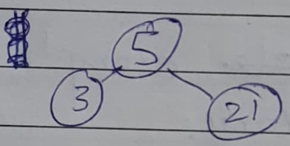
Q> Construct a B-tree of order 3

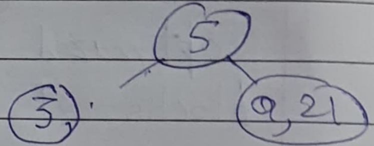
5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8

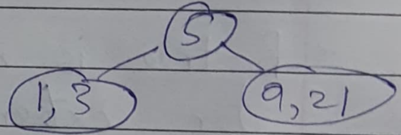
max keys = 2

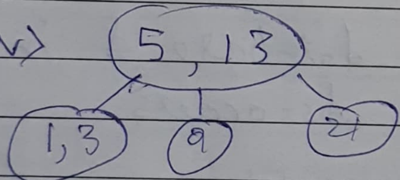
i>


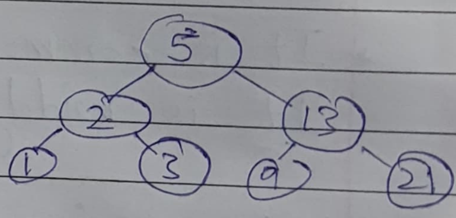
ii>


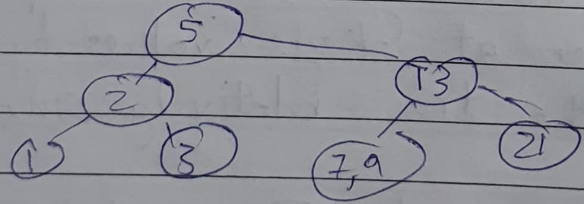
iii>


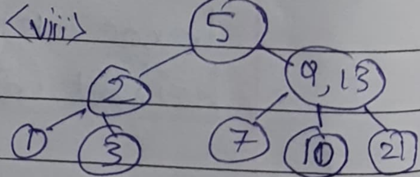
(iv>


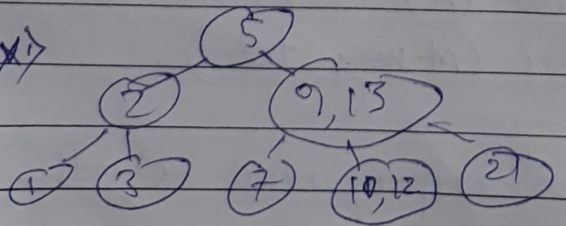
(v>


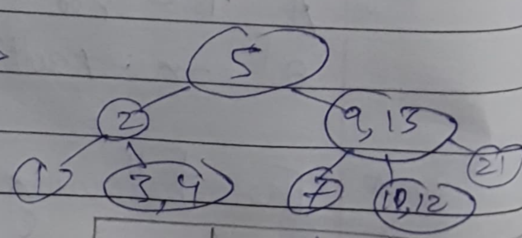
v>


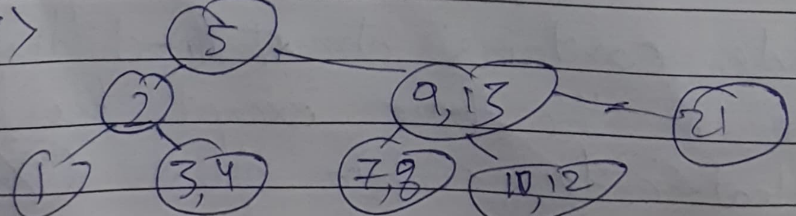
(vi)


(vii)


<viii>


<ix>


<x>


<xi>


	min	max
key	$\lceil \frac{m}{2} \rceil - 1 = 1$	$m-1=2$
child	$\lceil \frac{m}{2} \rceil = 2$	$m=3$

Q) Make B-Tree for 1 to 10 with order 3 & 5

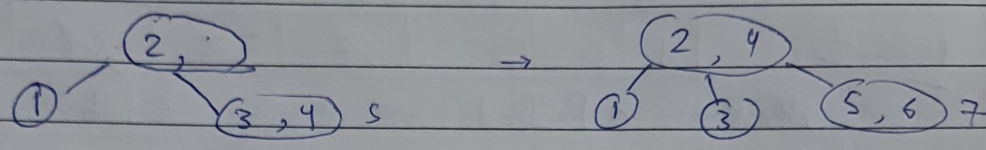
For order 3, $m = 3$

$Key = 3 - 1 = 2$

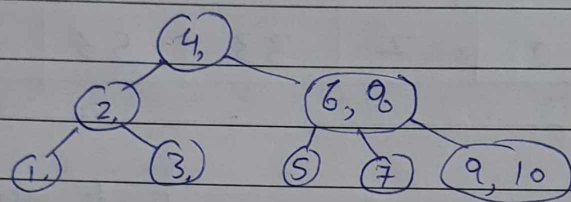
$mid = \left\lceil \frac{3}{2} \right\rceil = 2^{nd}$ element

i) (1, 2) 3

ii)



iii)



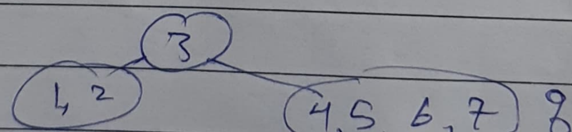
For order 5, $m = 5$

$Key = 5 - 1 = 4$

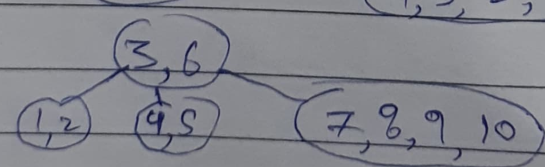
$mid = \left\lceil \frac{5}{2} \right\rceil = 3^{rd}$ element

i) (1, 2, 3, 4) 5

ii)

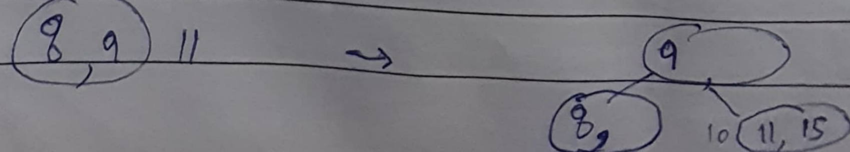


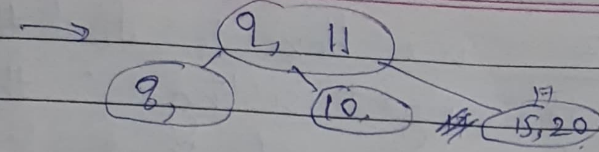
iii)

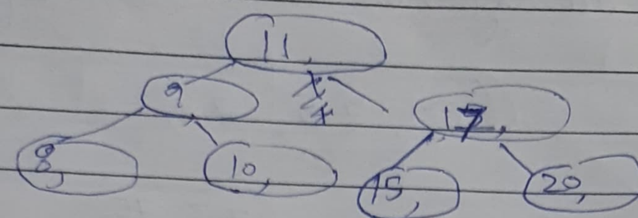


Q) Construct a B-Tree of order 3 for
8, 9, 11, 15, 10, 20, 17

$m = 3$, $m - 1 = 2$, $\left\lceil \frac{m}{2} \right\rceil = 2^{nd}$ element

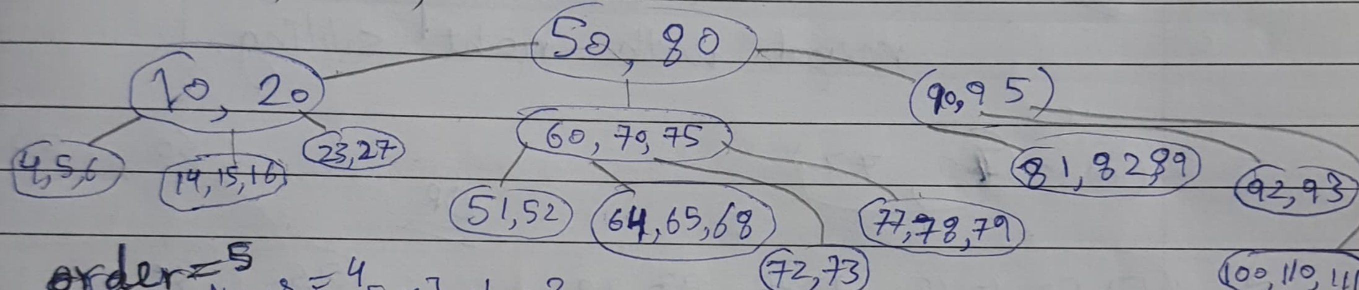






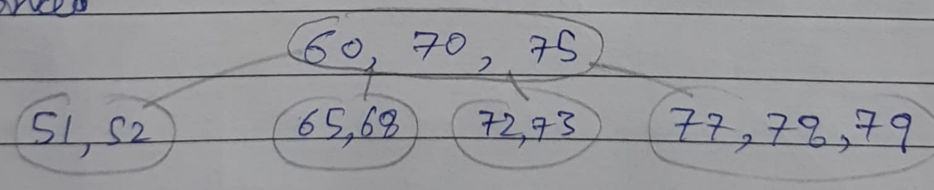
Deletion in B-tree

Q> Delete following elements from B-tree
64, 23, 72, 65, 20, 70, 95, 77, 80, 100, 6, 27, 60, 16, 50,



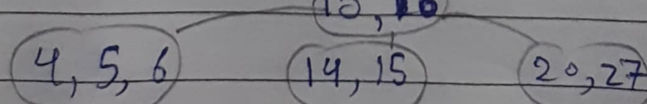
order = 5
max keys = 4
min keys = $\lceil \frac{m}{2} \rceil - 1 = 2$

~~Case~~ i > 64 can be deleted normally because it does not violate any properties, tree remains balanced



ii > ~~23~~ Deletion of 23 will lead to 27 being the only key in the node but to maintain two minimum keys we will ~~bor~~ find inorder predecessor value in its left sibling (16) & will shift it (16) to the root & bring root (20) ~~to~~ inplace of 23.

inorder predecessor from left



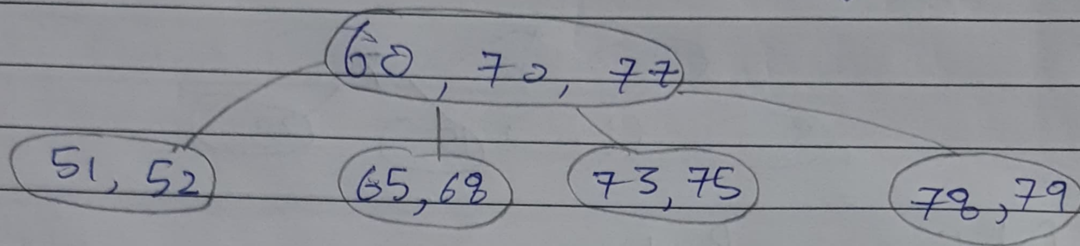
Jab left aur right sibling elem nahi de sakte tab Parent dono children ko merge kar do ~~dete~~

Date

Page

inorder successor in right node

iii) To ~~delete~~ delete 72 we have to find the inorder successor in the right sibling ∵ the left sibling already has min ~~node~~ keys



min keys in both left & right sibling = merging

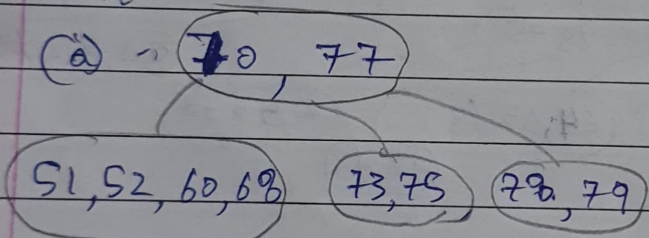
iv) On Deleting 65, either

(a) 60 will come in place of 65 & will get merged with left sibling

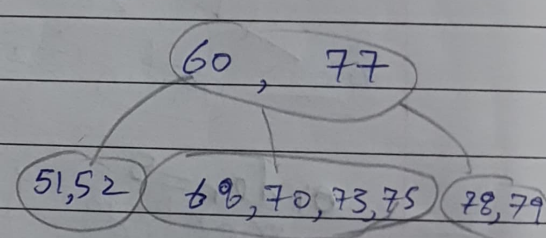
or (b) 70 will come in place of 65 & will get merged with right sibling.

i.e

(a) ∴



OR



Degree = minimum no. of children

RGPV → insertion only

Competitive exam → deletion

Date

Page

[For competitive exams]

Q> Consider a B-tree with the key size 10 bytes, block size 512 bytes, data pointer is of size 8 bytes & block pointer is of 5 bytes. Find the order of the tree.

order = m

~~max~~ keys = m − 1

Key size = 10 Bytes

block ptr = 5 bytes

block size = 512 bytes

data pointer = 8 bytes

$$m(5) + (m-1)(10) + (m-1)(8) \leq 512$$

$$5m + (m-1)(18) \leq 512$$

$$5m + 18m - 18 \leq 512$$

$$23m \leq 530$$

$$m \leq \frac{530}{23} = 23.04$$

order = $\boxed{m \simeq 23.}$

min ~~Keys~~ child node = $\left\lceil \frac{23}{2} \right\rceil = 12$

B tree grows Depth wise → used in secondary storage
B+ tree grows Breadth wise → used in DBMS & file system

Date
Page

B+ Tree

* B+ tree is an extension of B-tree which allows efficient insertion, deletion & searching operation.

* In B-tree, keys & records both can be stored in the internal as well as the leaf nodes. whereas, in B+ tree the records can only be stored in the leaf node while the internal nodes can only store the key values.

* The leaf nodes of B+ tree are linked together in the form of singly linked list to make the search queries more efficient.

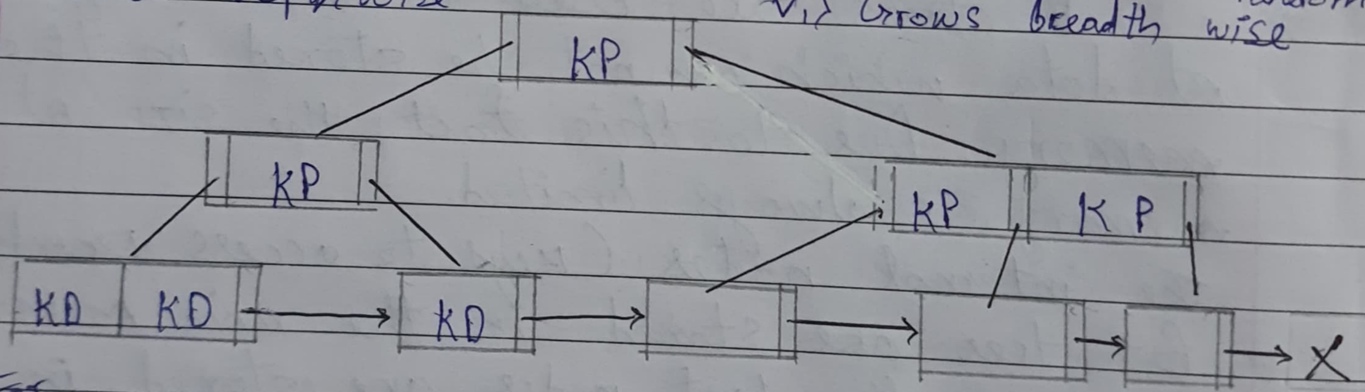
⇒ B+ trees are used to store large amount of data which can not be stored in the main memory. Due to this fact, the size of the main memory is always limited
The internal nodes (keys to access records) of the
* B+ tree are stored in the main memory whereas the leaf nodes are stored in the secondary memory.

* The internal nodes of B+ tree are often called as index node.

Difference between B tree & B+ tree

B tree	B+ tree
Data is stored in the leaf	Data is stored only in leaf

B tree	B+ tree
as well as in the internal nodes.	node
ii> No redundant search key is present in B tree.	Redundant keys may be present
iii> Leaf nodes are not linked together.	iii. Leaf nodes are linked together.
iv> Searching is slower and deletion is complicated and insertion is easy.	Searching is faster & deletion is easy because we have to delete from the leaf node.
v> Only for random searching	v> Sequential as well as random
vi> Grows depthwise	vi> Grows breadth wise



For Internal nodes,

$$m * \text{blockptr} + (m-1) * \text{keyptr} \leq \text{blocksize}$$

For leaf nodes,

Let no. of keyptr = no. of dataptr = y

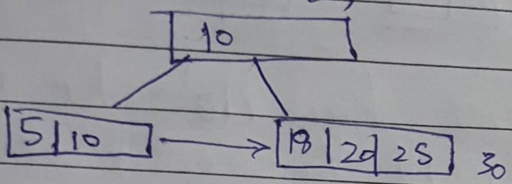
$$y * \text{keyptr} + y * \text{dataptr} + \text{blockptr} \leq \text{block size}$$

Q> ~~I~~ Insert in B+ tree
20, 10, 5, 18, 25, 30, 20, 21
order = 4

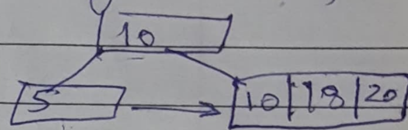
max ~~child~~ key = 3, min child = 2, min keys = 1

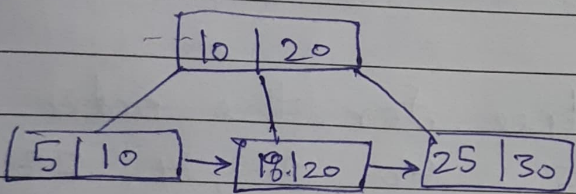
i> | 5 | 10 | 20 |
18

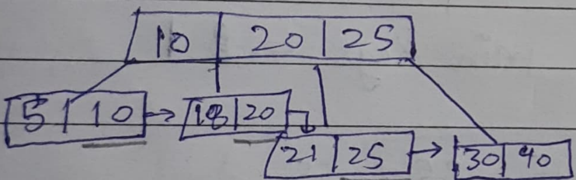
If left biased,



If right biased,



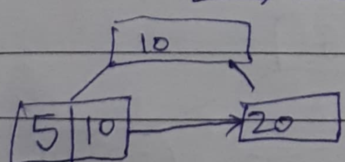


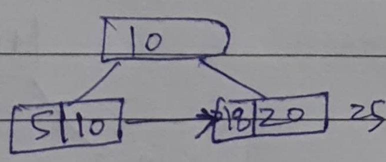


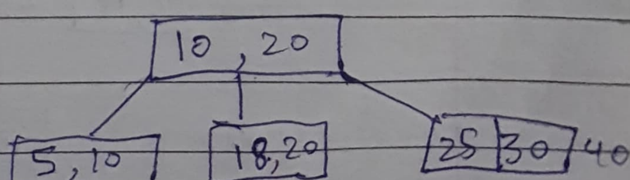
data Value < key Value

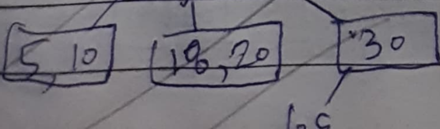
Same for order 3
key = 2, mid = 2

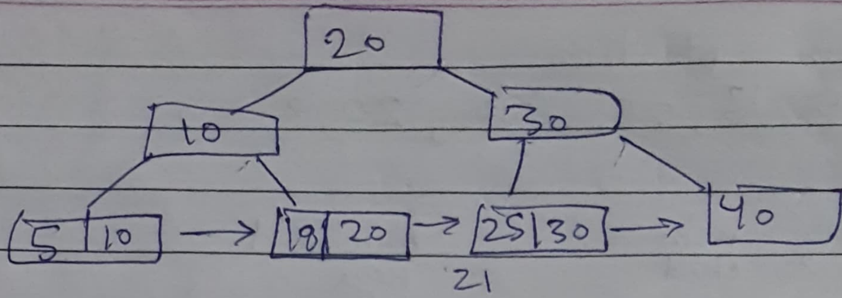
5 | 10, 20

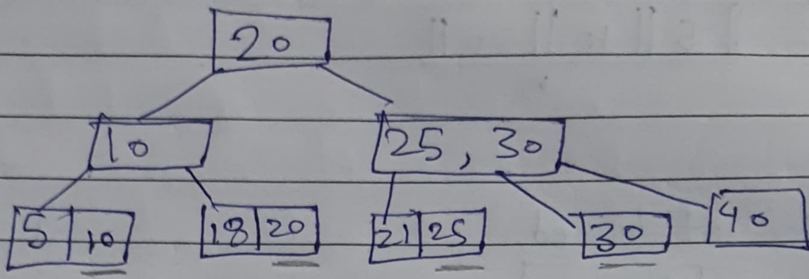








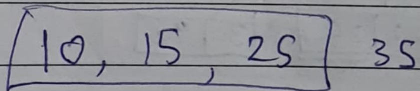


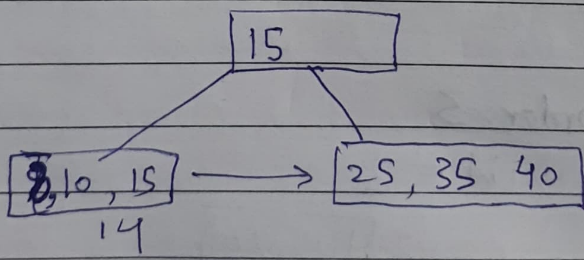


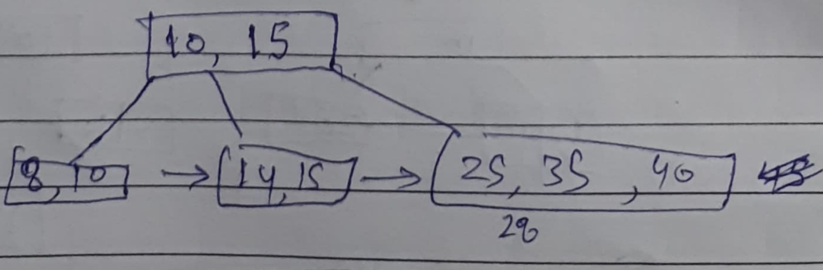
Left biasing mae keys ~~node~~ ke right side mae hota hai & right biasing mae node ke right mae hoti hai.

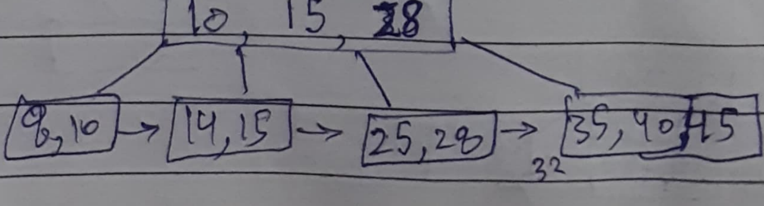
Q> Construct a B+ tree for the order 4
10, 25, 15, 35, 40, 8, 14, 28, 45, 32
3, 16

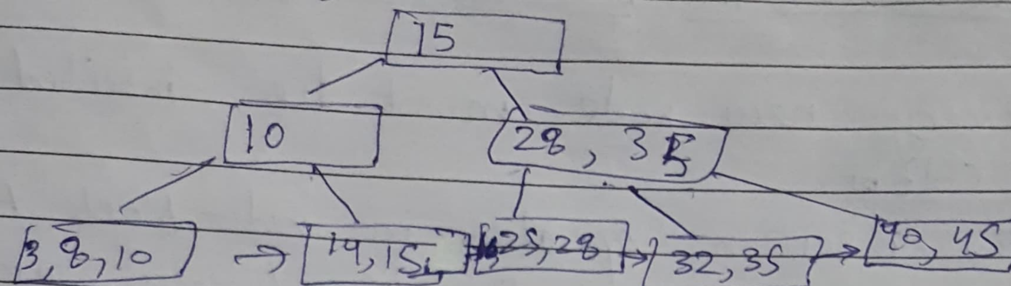
Sol max keys = 3, min keys = 1, mid = 2











Red - Black Trees

It is a kind of self balancing binary search tree. Each node of binary tree has an extra bit & that bit is often interpreted as the colour (red or black) of the node. ~~These~~
These colour bits are used to ensure the tree remains approximately balanced during the insertion & deletion

Properties of Red - Black tree :-

i) Red black tree must be binary search tree.
ii) A root node must be ~~inserted noo with the colour red~~ coloured black.
iii) Every ~~node~~ new node must be inserted with red colour.
iv) The children of red colour node must be coloured black (There should not be 2 consecutive red node)
v) In all the path of the tree there must be same number of black colour node.

memory → from mal indexing → AVL
memory
→ Red black →
less rotations → less data

Date
Page

Insertion in Red-Black tree

1> Every new node must be inserted with the red colour.

2> The insertion operation in red-black tree is similar to insertion operation in binary Search tree, but it is inserted with a colour property.

3> After every insertion operation we need to check all the properties of the red-black tree.

4> If all the properties are satisfied then we go to next operation otherwise we need to perform the following operation to make it red-black tree.

Operations :-

* Recolouring
* Rotation followed by recolouring

The insertion operation in red-black tree is performed by the following steps:

i> Check whether the tree is empty.

ii> If the tree is empty, then insert the new node as the root node with the colour black & exit from operation.

iii> If the tree is not empty, then insert the new node as leaf node with red colour.

iv> If the parent of new node is black then exit from the operation.

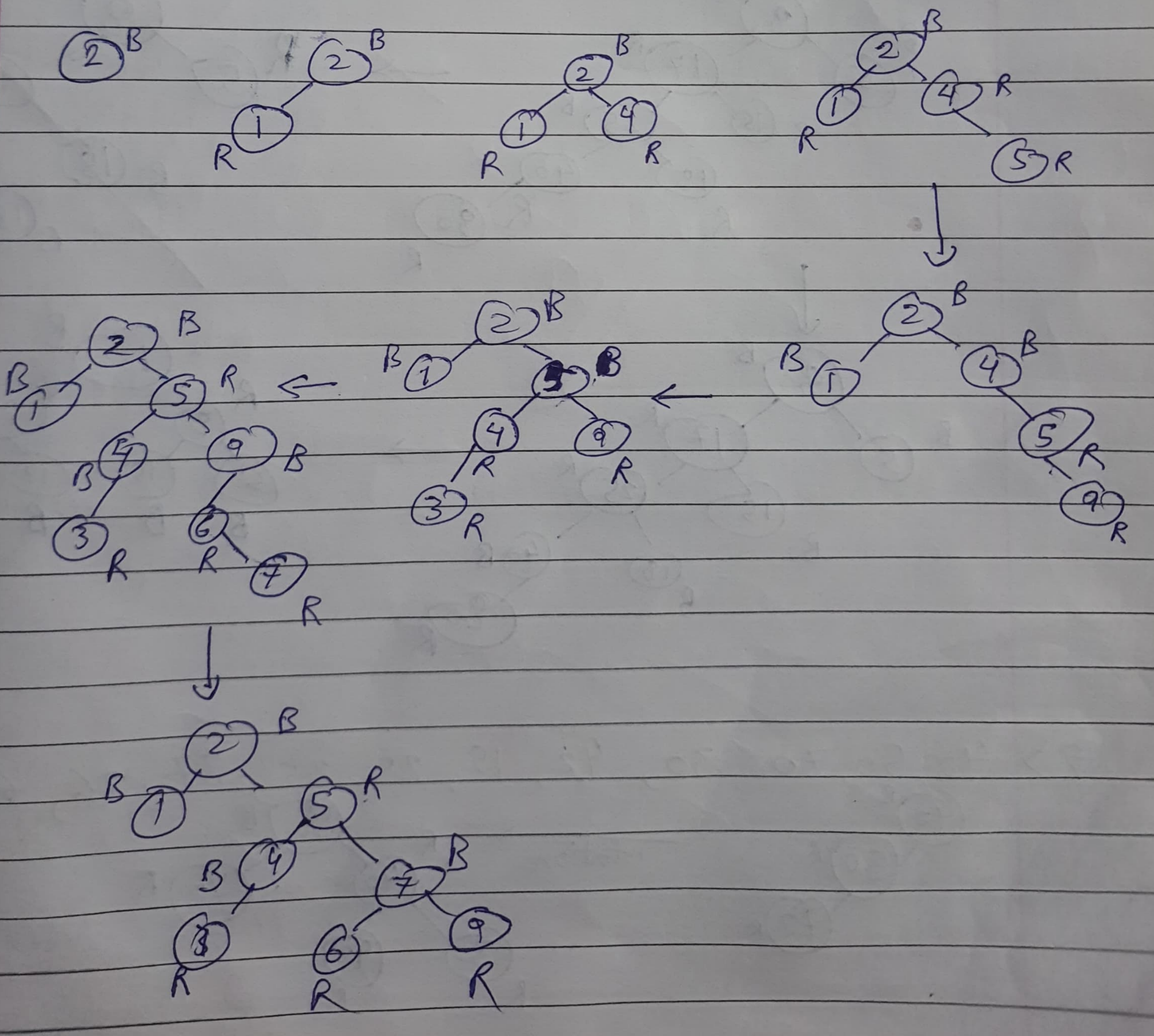
v> If the parent of the new node is Red, then check the colour of parent node sibling of the new node.

vi) If it is Black or NULL node then make suitable rotation & recolour.

vii) If it is red colour, then perform recolour & recheck it, repeat the same until the tree becomes red-black tree.

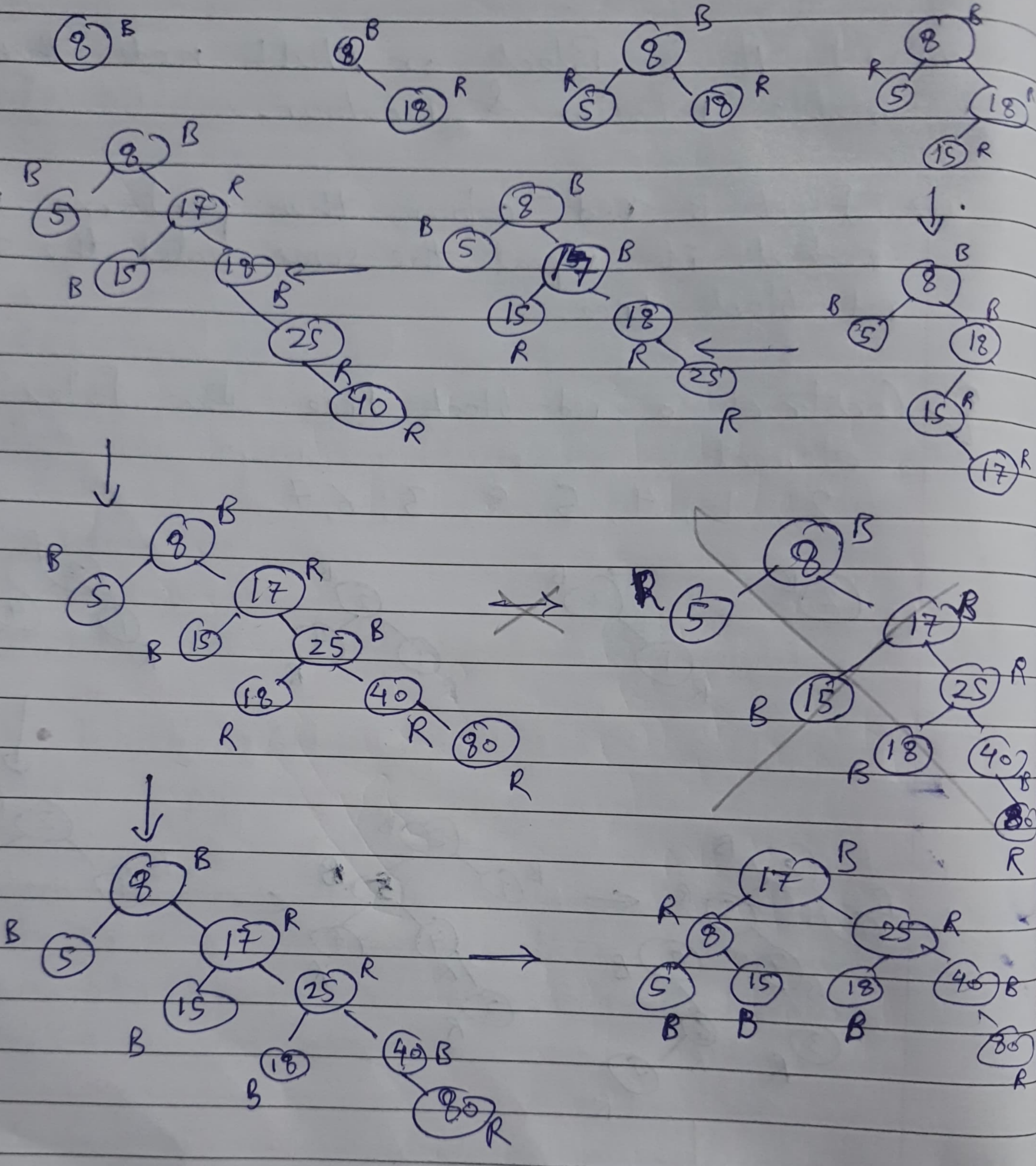
Q) Construct a red-black tree for following elements:

2, 1, 4, 5, 9, 3, 6, 7

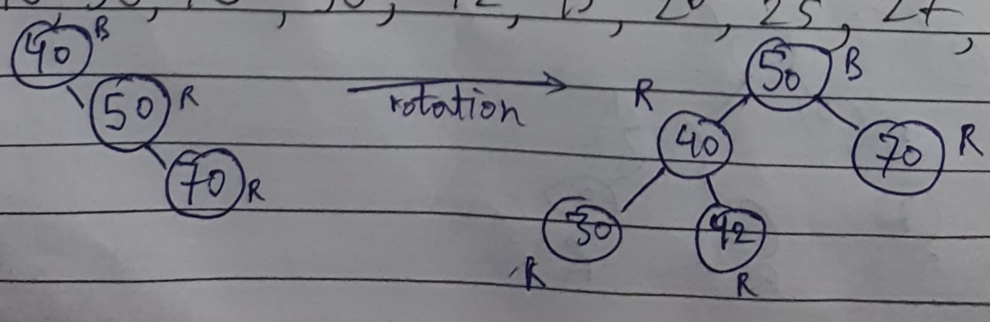


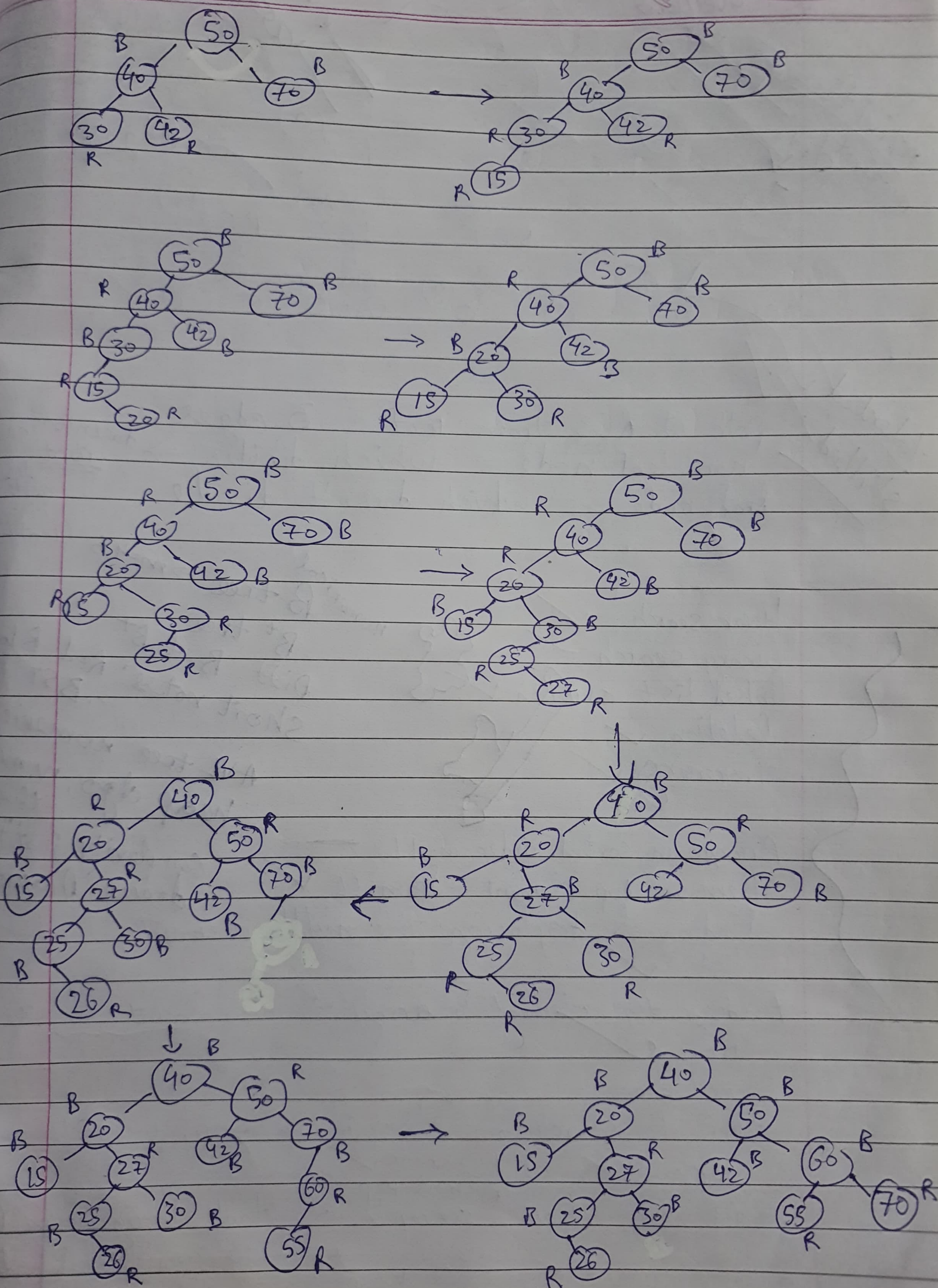
Make Red-Black tree

8, 18, 5, 15, 17, 25, 40, 80



?> 40, 50, 70, 30, 42, 15, 20, 25, 27, 26, 60, 55





Following is the detailed algo. The algo has mainly two cases depending on the colour parent's sibling (uncle)

* If the uncle's colour is red, we do recolouring

* If the uncle's colour is black, we do rotation & recolouring.

* Colour of the null node is considered ~~&~~ as black.

Let x be newly inserted node.

i> Perform standard Binary Search Tree insertion & make the colour of newly inserted node as red.

ii> If x is the root node, change the colour of x as black colour.

iii> Do following, if the colour of x's parent is not black or x is not root:

Case a> If x's uncle's colour is red and grandparent's colour is black:

→ Change the colour of parent & uncle as black
→ Do not change colour of grandparent if it is the root node otherwise change it to red.
→ Change the colour of parent & grandparent & repeat the step ii & iii for a new node x.

Case b> If x's uncle is black or null then do four configurations for the x, x's parent & x's grandparent
→ Left-Left case: P is Left child of g &

x is the left child of P.

→ Right-Right case: P is RC of g & x is RC of P

→ Left-right case: P is LC of g & x is RC
P

→ right-left case: P is RC of g & x is LC of P

[rotations perform karoke dikane hain saare rotation mae]

Unit-5
Sorting

* The things which is different from int, float, char and is user defined in nature is known as non-primitive data structure

* Data structures are basically a blue print / template implemen-ted by using an algorithm in a particular language.

Sorting - The process of arrangement of data in a systematic order is called sorting.
By sorting the data, it is easier to search through it quickly & easily. The example of sorting is dictionary.

Sorting algorithms - It is a just a series of order or instructions. In this an array is an input on which the sorting algos perform operations to give out a sorted array.

Importance of sorting in data structures

There are different types of sorting methods in data structures. Sorting actually provide user with several advantages. ~~eg - Whe~~
eg - When we perform sorting an elements, many complications like to find the minimum or maximum number, k^{th} smallest or largest element in an array get automatically simplified.

Furthermore, sorting also provide with many algorithmic solutions same of which might be some of might include ~~itera~~ iterative, recursion and divide & conquer.

One of the major advantage of sorting is its time complexity. The ultimate goal is to solve any kind of complex problem within the minimum amount of time. That is why different types of sorting strategies come into existence.
It not only saves time but also provides you with the right solution.

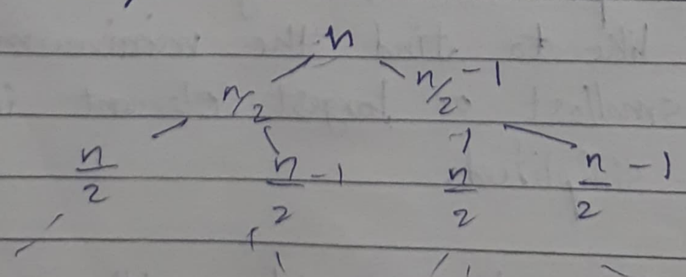
Types of sorting

Sorting is basically of two types :-

Internal sorting = * Performed in primary memory. * Input data is such that can be adjusted in main memory at once.
* Data set is small.

External sorting - * Performed in auxillary memory.
* Data set is bigger.
* If the input data is such that it can not be adjusted in main memory at once, it needs to be ~~stor~~ sorted in secondary memory (Floppy disk, hard disk, pendrive, SSD, CD, DVD or any memory storage devices)

Quick Sort / Partition Sort
* Divide & conquer
* stable



Recurrence relation = $T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + O(n)$

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \quad (1)$$

Put $n = \frac{n}{2}, \frac{n}{4}, \frac{n}{8}$ ---- $\frac{n}{}$

Sorting

1> Bubble Sort
2> Insertion Sort
3> Selection Sort
4> Quick Sort
5> Merge Sort
6> Radix Sort
7> Shell Sort
8> Heap Sort

Bubble Sort → Compare two index at a time

Q> 5, 3, 8, 4, 6

Pass-1: swap

0	5	3	3	3	3
1	3	5	5	5	5
2	8	8	8	4	4
3	4	4	4	8	6
4	6	6	6	6	8

Pass 2:

0	3	3	3	3
1	5	5	4	4
2	4	4	5	5
3	6	6	6	6
4	8	8	8	8
	a[0] & a[1] no swap	a[1] & a[2] swap	a[2] & a[3] no swap	Sorted array

Q> 20, 35, -15, 7, -55, 1, -22

Pass 1

0	20	20	20	20	20	20	20
1	35	35	-15	-15	-15	-15	-15
2	-15	-15	35	7	7	7	7
3	7	7	7	35	-55	-55	-55
4	-55	-55	-55	-55	35	1	1
5	1	1	1	1	1	35	-22
6	-22	-22	-22	-22	-22	-22	35
	a[0] & a[1] no swap	a[1] & a[2] swap	a[2] & a[3] swap	a[3] & a[4] swap	a[4] & a[5] swap	a[5] & a[6] swap	

Pass 2

0	20	-15	-15	-15	-15	-15
1	-15	20	7	7	7	7
2	7	7	20	-55	-55	-55
3	-55	-55	-55	20	1	1
4	1	1	1	1	20	-22
5	-22	-22	-22	-22	-22	20
6	35	35	35	35	35	35

Pass 3

0	-15	-15	-15	-15	-15
1	7	7	-55	-55	-55
2	-55	-55	7	1	1
3	1	1	1	7	-22
4	-22	-22	-22	-22	7
5	20	20	20	20	20
6	35	35	35	35	35

Pass 4

0	-15	-55	-55	-55
1	-55	-15	-15	-15
2	1	1	1	-22
3	-22	-22	-22	1
4	7	7	7	7
5	20	20	20	20
6	35	35	35	35

Pass 5

0	-55	-55	-55
1	-15	-15	-22
2	-22	-22	-15
3	1	1	1
4	7	7	7
5	20	20	20
6	35	35	35

Sorted Array

Insertion Sort → Compare an index with all previous indices

Q) 20, 35, -15, 7, -55, 1, -22

Pass 1

0	1	2	3	4	5	6
20	35	-15	7	-55	1	-22

compare a[1] with a[0] → No swapping

Pass 2

20	35	-15	7	-55	1	-22

a[2] with a[1] → swap

20	-15	35	7	-55	1	-22

a[1] with a[0] → swap

-15	20	35	7	-55	1	-22

Pass 3

-15	20	35	7	-55	1	-22

a[3] with a[2] → swap

-15	20	7	35	-55	1	-22

a[2] with a[1] → swap

-15	7	20	35	-55	1	-22

a[1] with a[0] → No Swap

Pass 4

-15	7	20	35	-55	1	-22

a[4] with a[3] → swap

-15	7	20	-55	35	1	-22

a[3] with a[2] → swap

-15	7	-55	20	35	1	-22

a[2] with a[1] → swap

-15	-55	7	20	35	1	-22

a[1] with a[0] → swap

-55	-15	7	20	35	1	-22

Pass 5

-55	-15	7	20	35	1	-22

a[5] with a[2] → Swap

-55	-15	1	7	20	35	-22

Pass 6

-55	-15	1	7	20	35	-22

a[6] with a[~~1~~] → swap

-55	-22	-15	1	7	20	35

Sorted Array

Selection Sort → Select minimum in each pass & move it to front or maximum to back of array

Q) 33, 7, 2, 0, 1, 98, 87, 56

Pass 1

0	1	2	3	4	5	6	7
33	7	2	0	1	98	87	56

Max → 98 a[5] → swap with 56 a[7]

33	7	2	0	1	56	87	98

Pass 2

33	7	2	0	1	56	87	~~98~~

Max → 87 → already at right position

Pass 3

33	7	2	0	1	56	~~87~~	~~98~~

Max → 56 → already at right position

Pass 4

33	7	2	0	1	~~56~~	~~87~~	~~98~~

Max → 33 a[0] → Swap with 1 a[4]

1	7	2	0	33	56	87	98

Pass 5

1	7	2	0	33	56	87	98

Max → a[1] = 7 → Swap with a[3] = 0

1	0	2	7	33	56	87	98

Pass 6

1	0	2	7	33	56	87	98

Max → 2 → already at correct position

Pass 7

1	0	2	7	33	56	87	98

Max → a[0] = 1 → Swap with a[1] = 0

0	1	2	7	33	56	87	98

Quick Sort / Partition Sort → Divide array into Two equal or unequal parts

→ Select a pivot → 1st index or mid index or last index

→ look for smallest from right & largest from left

Best ~~Worst~~ case → $O(n \log n)$

worst case → $O(n^2)$ when array is already sorted.

$T(n-1) \to T(n)$

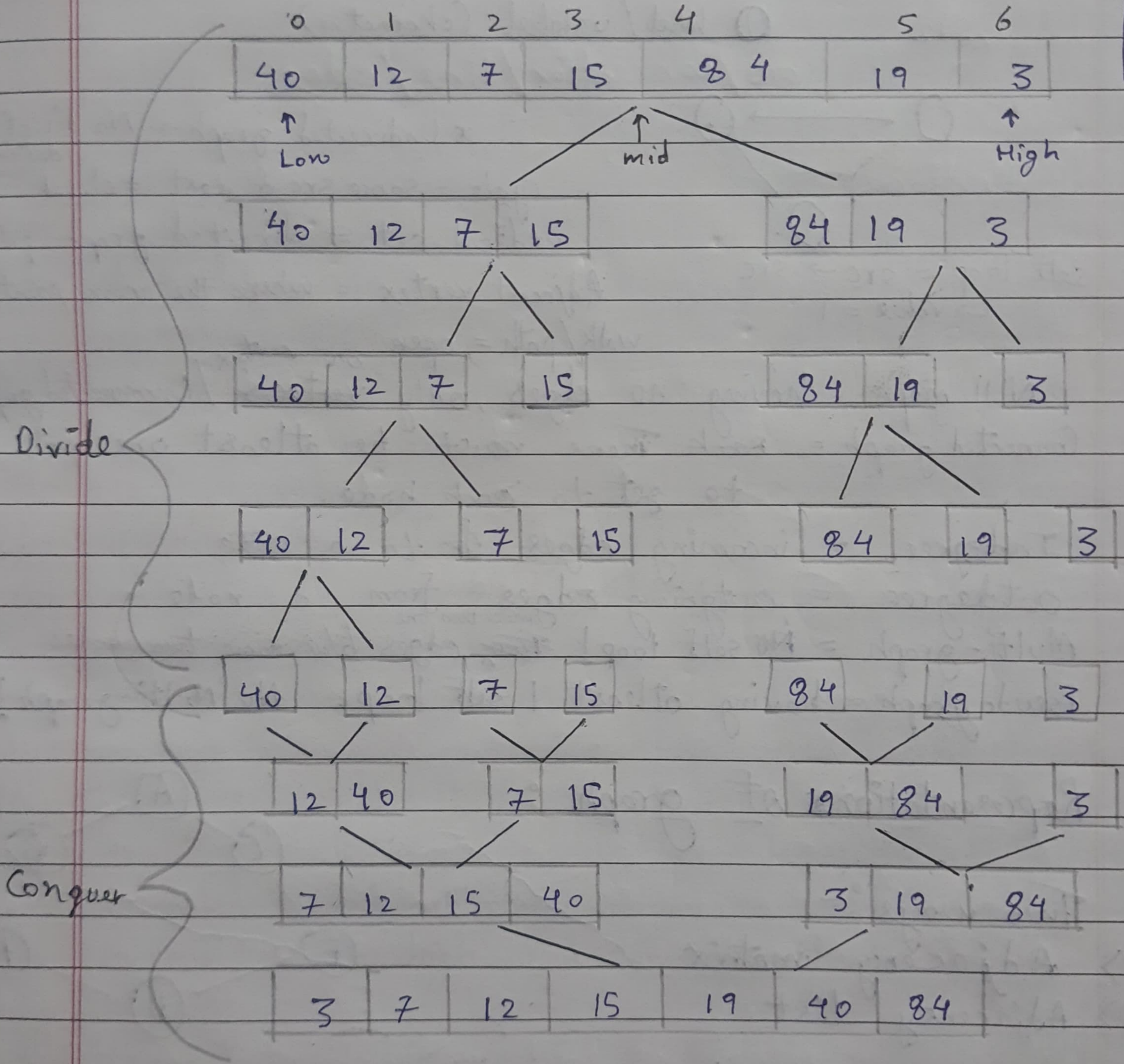
Merge Sort

* Two ways → iterative → we have to select slab value ~~for merging~~

→ Recursive → Divide & Conquer.

$mid = \frac{low + high}{2}$

$\left\lceil \frac{6}{2} \right\rceil = 3$



Radix Sort / Bucket Sort

It is a non-comparison based sorting technique

$O(n \times d)$

↳ digits

Sorting Programs :-

```
void bubbleSort (int *a , int n ){
        int i, j;
    for (i=0 ; i<n-1 ; i++){
            for (j=0 ; j<n-i-1 ; j++){
                    if (a[j] > a[j+1]){
                            temp = a[j];
                            a[j] = a[j+1];
                            a[j+1] = temp;
                    }
            }
    }
}

void selectionSort (int *a , int n) {
        int i, j, position;
    for (i=0 ; i<(n-1) ; i++){
        position = i;
        for (j= i+1 ; j<n ; j++){
            if (a[position] > a[j])
                position = j;
        }
        if (position != i) {
            temp = a[i];
            a[i] = a[position];
            a[position] = temp;
        }
    }
}
```

Algo - MergeSort (arr, low, high) {

if (low < high) {

mid = $\left\lfloor \frac{low + high}{2} \right\rfloor$

MergeSort (arr, low, mid);
MergeSort (arr, mid+1, high);
Merge (arr, low, mid, high);

}

}

Algo Merge (a, low, mid, high) {

i = low;
j = mid + 1;
k = low;
~~while (k ≤~~
~~if (a[i] ≤ a[j]) {~~
~~b[k] = a[i];~~
~~i++;~~

create two subarrays : left sub-array or a[i] ranges from 0 to mid+1

set mid+1 to infinity.

: create second subarray named as a[j] ranges from mid+2 to high and set high+1 to infinity.

Case 1 : While a[i] ≤ a[j] and i ≤ mid and j ≤ high then { b[k] = a[i];
i++; k++; }

Case 2 : Left subarray is not present so simply copy the elements from right ~~sb~~ subarray
if (i ≥ mid) { while (j ≤ high) { b[k] = a[j];
j++; k++; }}

Case 3: we ~~are~~ having only left subarray and the right subarray is absent so simply copy the elements from left subarray.

```
if (j ≥ high) {
        while (i ≤ mid){
                b[k] = a[i];
                i++; j++;
        }
}
```

Lastly shift all the elements from the temporary array to the actual array (a) i.e

```
For k := low to high {
        a[i] = b[k];
}
```

Time Complexity of merge sort:

For all three cases, the complexity will be $O(n \log n)$ where n represents total numbers of elements in array.

It is not consider as a in-place sorting technique because it requires extra memory for ~~so~~ storing the data elements.

It is considered as an external sorting technique & that's why it is beneficial for the larger dataset.

Q) The average number of comparisons performed by merge sort in sorting two sorted list of length 2 is how much? $\Rightarrow n \log n = 4 \log(4) = 4(2) = 8$

$\underset{\text{max comparison}}{8}$ $= \frac{8}{3} = 2.667$

Best sorting technique → ① insertion / online
→ Sorts as the data comes ~~in~~
rather than first making an array

② Quick Sort > Merge Sort
↓ executes in primary memory
↳ executes in secondary memory.

Slowest technique → Selection Sort

Graph

↓

(vertices, Edges)

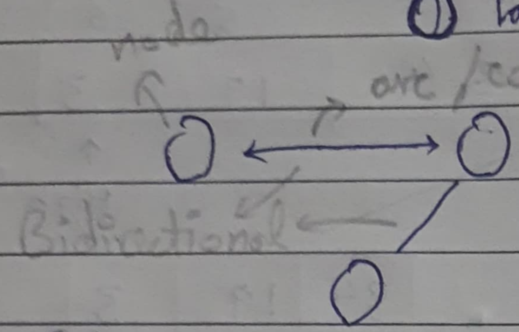
↓

node = data / info

↓

O label / unlabel (character)

* non-primitive
* non-linear
* closed loop



* Undirected graph → No directions defined

cycle = same src & dest = closed

ordered set = directed graph path

self loop = src → src
↳ value = 1

Adjacent vertex = where the arrow points

walk/path = open

Null graph = having no edges, only ~~vertices~~ one vertex / Unconnected graph

Connected graph = ~~Each~~ ~~There~~ must be atleast one path to get to each node.

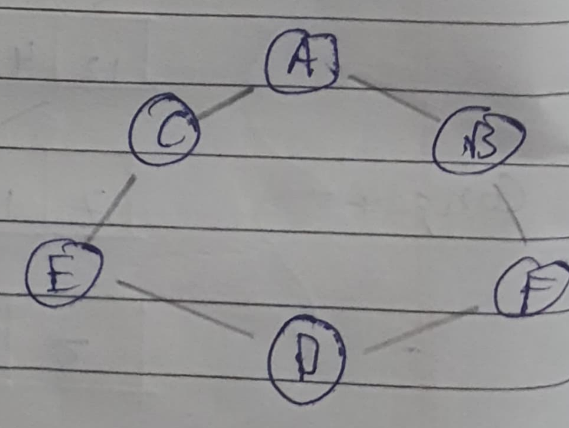
Indegree = incoming edges to a node

outdegree = outgoing edges from a node

Multi-graph = No self loop! ~~Two~~ more than one edges b/w same two nodes

Pseudograph = having atleast 1 self loop, No Multi-graph!

Representations of graph :



Two ways :-

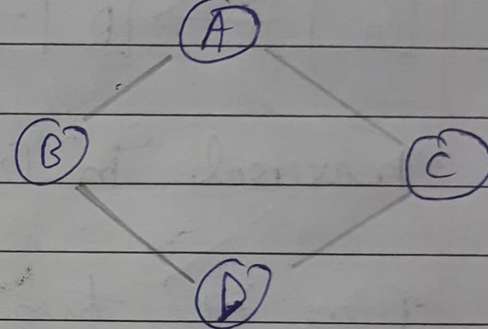
1> Adjacency matrix

2> Adjacency list

Graph Representation

① Adjacency Matrix

$$
\begin{array}{c|c}
 & \begin{array}{cccc} A & B & C & D \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \end{array} &
\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}
\end{array}
$$



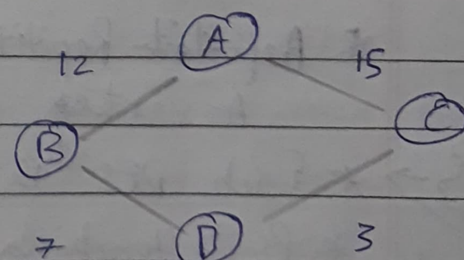
$A_{ij} = 1$

row   col

$A_{ii} = 0$

eg → weighted graph

$$
\begin{array}{c|c}
 & \begin{array}{cccc} A & B & C & D \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \end{array} &
\begin{bmatrix} 0 & 12 & 15 & 0 \\ 12 & 0 & 0 & 7 \\ 15 & 0 & 0 & 3 \\ 0 & 12 & 15 & 0 \end{bmatrix}
\end{array}
$$



② Adjacency List representation

→ linked list representation is done for each & every vertex of the graph

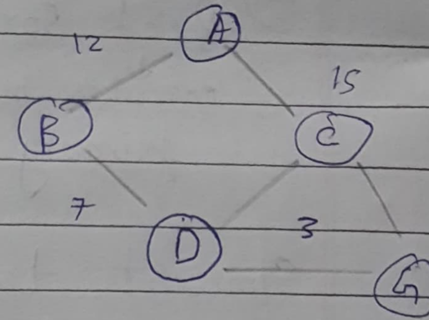
A → B → C | NULL

B → D → A | NULL

C → A → D | ~~NULL~~   G | NULL

D → B → C | ~~NULL~~ → G | NULL

G → D → C | NULL



* Traversal in Graph

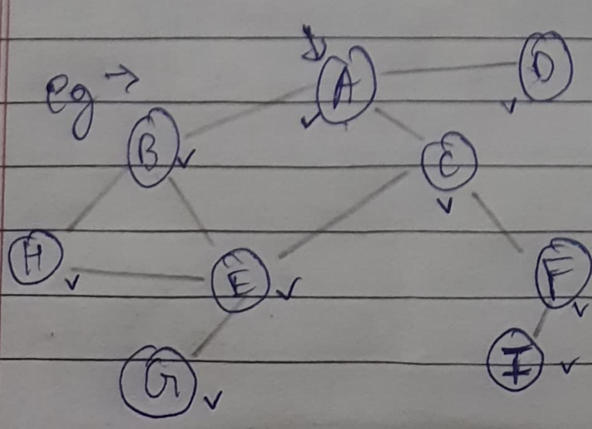
There are two ways to traverse a graph :-

Blind Search / Uninformed Search
① DFS - Depth First Search - Stack data structure
② BFS - Breadth First Search - Queue data structure

* ~~Repetit~~ Repitition of vertex is not allowed in traversal.
A vertex can only be visited once.

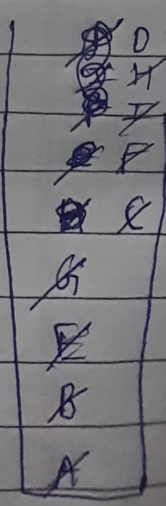
DFS → * Each DFS is unique in nature
* Back tracking is used in DFS

Eg →



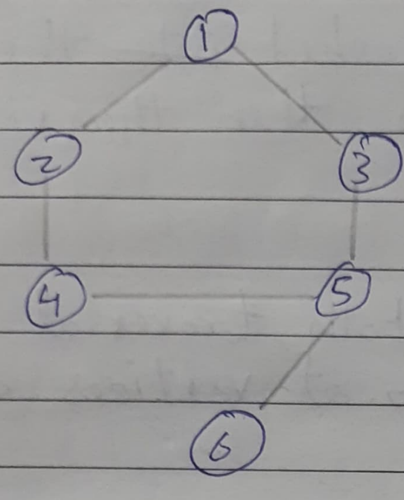
⇒ ~~A B E G H B C F I D~~

⇒ A B E G C F I H D



Algorithm:

eg →



Visited (array)

initial →	0	0	0	0	0	0
	1	2	3	4	5	6
final	1	1	1	1	1	1

```
Algo DFS (v) {
    visited [v] = 1;
    for each adjacent vertex u to v{
        if (visited[u]==0)
            { DFS (u);}
    }
}
```

Difference b/w DFS & BFS

BFS → ① BFS stands for Breadth First Search and it is a vertex based technique for finding the shortest path in the graph

It uses the queue datastructure that follows the pricipal of FIFO.

In BFS, One vertex is selected at a time when it is visited and marked as visited.
Then, its adjacent ~~vertex~~ vertices are visited and stored in queue.

It is slower than DFS but it guarantees that it will always be connected and we will reach the goal

DFS → It is an edge-based technique & it uses the stack data structure and is performed in two stages

i> Visited vertices are ~~to~~ pushed into the stack.
ii> If there are no vertices then the visited vertices are popped out.

⇒ The time complexity for both traversal is $O(V+E)$ where V is no. of vertices & E i.e. no. of edges.

(2) ~~In~~ BFS doesn't have concept of back tracking while DFS works on technique of backtracking

DFS may be connected or unconnected & it does not guarantee that ~~will~~ we will reach to goal state i.e they might get fixed in a loop.

(3) BFS is used to find the shortest path,
DFS is used for the topological network (LAN, MAN, WAN)

(4) BFS requires more memory while DFS requires less memory

(5) When the target is close to the source vertex then BFS performs better.
When the target is far away from the source then DFS is preferable

Date
Page

```
Algo BFS (v) {
    u = v;
    visited [v] = 1;
    for each adjacent vertex w to u {
            if (visited [w] = = 0 ) {
                visited [w] = 1
                Insert w in queue;
            }
        }
    if (queue is empty) return;
    else Dequeue v from queue;
}
```

Note → repeat until more nodes are present, No recursive Call.

Minimum Spanning Tree

A spanning tree is the sub-graph of an undirected connected graph.

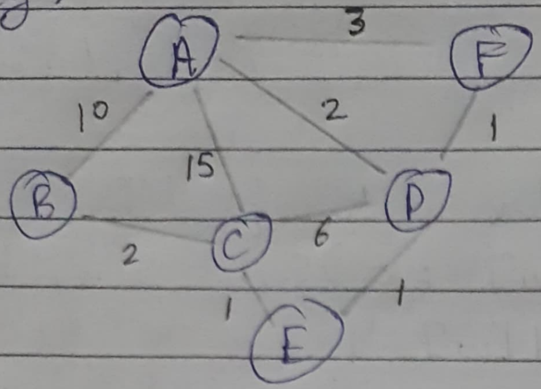
The minimum spanning tree can be defined as the spanning tree in which the sum of the weight of the edge ~~of~~ is minimum.

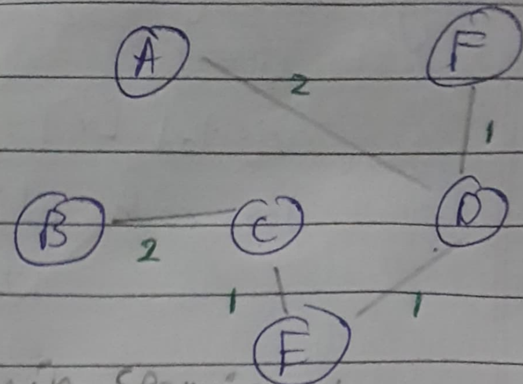
The weight of the spanning tree is the sum of the weight of the given edges of the spanning tree.

Two ways of making minimum spanning tree :-

i) Kruskal
ii) Prim's

eg →



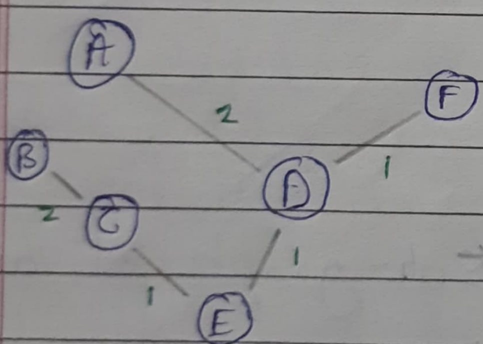


min spanning tree

⇒ Applying Kruskal, mincost = 0 + 1 + 1 + 1 + 2 + 2 = 7
↳ minimum edge wise

Note → All nodes should be present & connected in resultant graph.
→ Draw each step in exam, except discarded edges.

⇒ Applying Prim's, min cost = 0 + 2 + 1 + 1 + 1 + 2 = 7
A D F E C B



* All vertices should be traversed
→ minimum edge FROM current node

⇒ If the values are distinct in nature then we will get the same spanning tree for both the methods

But if the values are same then it might be possible that the spanning tree will differ but the cost remains the same.

Kruskal * Kruskal is a greedy algorithm in the graph theory that finds a minimum spanning tree for a connected weighted graph.

* It finds a sub-set of edges that forms a tree that includes every vertex of the graph where the total weight of all the edges in the tree is minimal.

* Kruskal algo addresses two problems :-
  i) It gives a ~~p~~ practical method for constructing a spanning subtree of minimal length

  ii) It gives a practical method for constructing an unbranched spanning tree of the minimum length.

Algorithm :-

1) Create a forest in such a way that every vertex of the graph is a separate tree. [Adjacency matrix]

2) Create a set of edges that contains all the edges of the graph. [Min heap]

3) Repeat the step 4 & 5 while the edge set is not empty & ~~forest~~ forest is not spanning tree.

4) Remove an edge from the edge set with the minimum weight

5) If the edge obtained in step no. 4 connect two different trees than add it to the forest for combining

two trees into one tree else discard the edge is it is forming the cycle.

Complexity of Kruskal Algo.

* It takes $O(E \log E)$ time in sorting of the edges where E is no. of edges & V is no. of vertices in the graph.

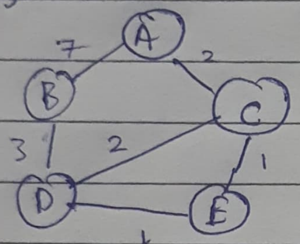
Further it iterates all the edges & runs a subroutine to find a cycle in the graph which is called union-find algorithm

The union-find algorithm requires $O(\log B)$ time and is applied after sorting of the edges. is completed

Dijastra's Algorithm

* It is a single source, shortest path, algorithm technique to find out the shortest route or the shortest distance b/w two vertices.

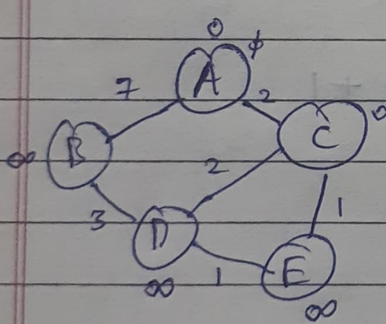
* In this technique the source vertex will be fixed and we have to calculate the shortest route from the source vertex.
* Dijastra's works on greedy technique.



$d(x, y) = d(x) + c(x, y) < d(y)$

Source dest

Mark (A) as ϕ and assign cost 0 to it.
Assign cost ∞ to every other node initially



(1) Source is A which is connected to B & C

$d(A, C) = d(A) + c(A, C) < d(C)$
$= 0 + 2 < \infty$
$= 2 < \infty$ $\because$ true

$\therefore$ Cost 2 is assigned to C

$d(A, B) = d(A) + c(A, B) \leq d(B)$
$= 0 + 7 < \infty$
$= 7 < \infty$ $\because$ true

$\therefore$ cost 7 is assigned to B

$\because$ $d(A, C) < d(A, B)$

$\therefore$ We will move from A to C i.e Path = A → C

Dijastra's

Given a weighted graph, the shortest path b/w two vertices is a path with the lowest possible sum of edge weight.

Comparison b/w Prim's & Dijastra's Algo

Dijastra's	Prim's
i) Dijastra's algo finds the shortest Path.	i) Finds the min spanning tree.
ii) Works on both directed & undirected graph	ii) Only works on undirected graph.
iii) May fail to compute the accurate result if there's atleast one negative edge.	iii) Can have negative edges
iv) In practice, Dijastra's algo is used when we want to save the time & fuel travelling from one point to another.	iv) Prim's is used when we want to minimize the material cost in constructing roads that connects multiple points to each other.

Application of Dijastra's Algo

i) It is used in IP routing to find the shortest path. For this OSPF is a linked-state routing protocol that is used to find the best path b/w the source & the destination router.

ii) It is widely used in routing protocols required by the router to update their forwarding table.

The algo provides the shortest cost path from source router to the other routers in the network

ii) Digital Mapping
Many times we try to find distance in google Maps from one city to another or from your location to the nearest desired location, this is possible thru Dijastra's Algo.

iii) Robotic Path
Nowadays, drones & robots have come into existence.
- Some of them are manual & Some are automated.

The drones which are automated are used to dilever packages to specific locations or used for a task that are loaded with this algo module so that when source & dest is known, the drone or robot moves in the ordered direction by following the shortest dist to keep delivering the packets in min amt of time.

* It works on BFS traversal i.e uses queue Data structure.

* Complexity → $O(V^2)$
V is no. of vertices

HASHING

Hashing is an approach to convert a key into an integer within a limited range.

This key to address transformation is known as hashing function which maps the key space into an address space.

Thus the hash function produces a table address where the record may be located for the given key-value.

Ideally no two keys should be converted into the same address.
But there is no hash function that guarantees that this situation will never occur.
This situation is called as collision.

⇒ Collision
When the hash f^n produces the same address for the two keys then this situation is known as collision.

There are certain collision resolution techniques that are as follows:-

i) Open Hashing
ii) Close Hashing

⇒ Hash Table

A hash table is an array that is address ~~to~~ thru a hash function.

The hash table is divided into a no. of buckets & each bucket in turn capable of storing a no. of records.

Thus we can say that a bucket has no. of slots & each slot is capable of holding one record.

⇒ Difference b/w index function & Hash function

An index function gives the sequential location of an array while hash function takes a key & maps it to some index in an array.

⇒ Collision removing technique thru
① linear probing

Linear probing is a hash technique & it is known to be the easiest way to resolve any collisions in the hash tables.

* A sequential search can be performed to find any collision that occurs.

② <u>Double Hashing</u>

It uses two hash functions. The second hash function comes ~~the~~ into use when the first function causes a collision.

It provides an offset index to store the value.
The formula for this is

(First hash (key) + I * Second hash (key)) % size of the table

where I is the offset value. This value keeps incrementing until it finds an empty slot.

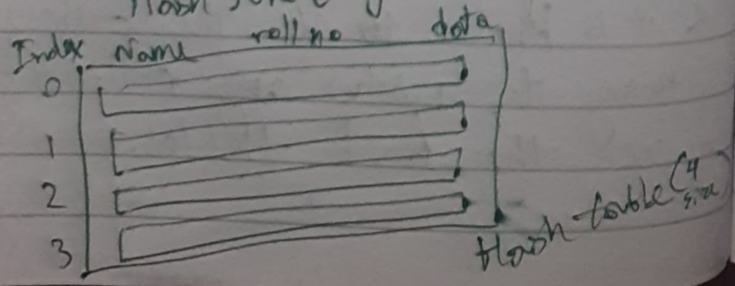
Hashing → kind of array ↳ custom arr or associative arr or structure type arr

↓

Hash table

(key + value) → into

Hash func (key) ⇒ Index



Hash function?

Key % table size = index

Linear probing eg → $1722 \% 10 = \boxed{2}$
$3572 \% 10 = \boxed{2}$ — Collision

* Hashing makes use of priority queue to store the records. if necessary

<u>Time Complexity for Hashing</u>

Constant TC → O(1)