20/5/2024

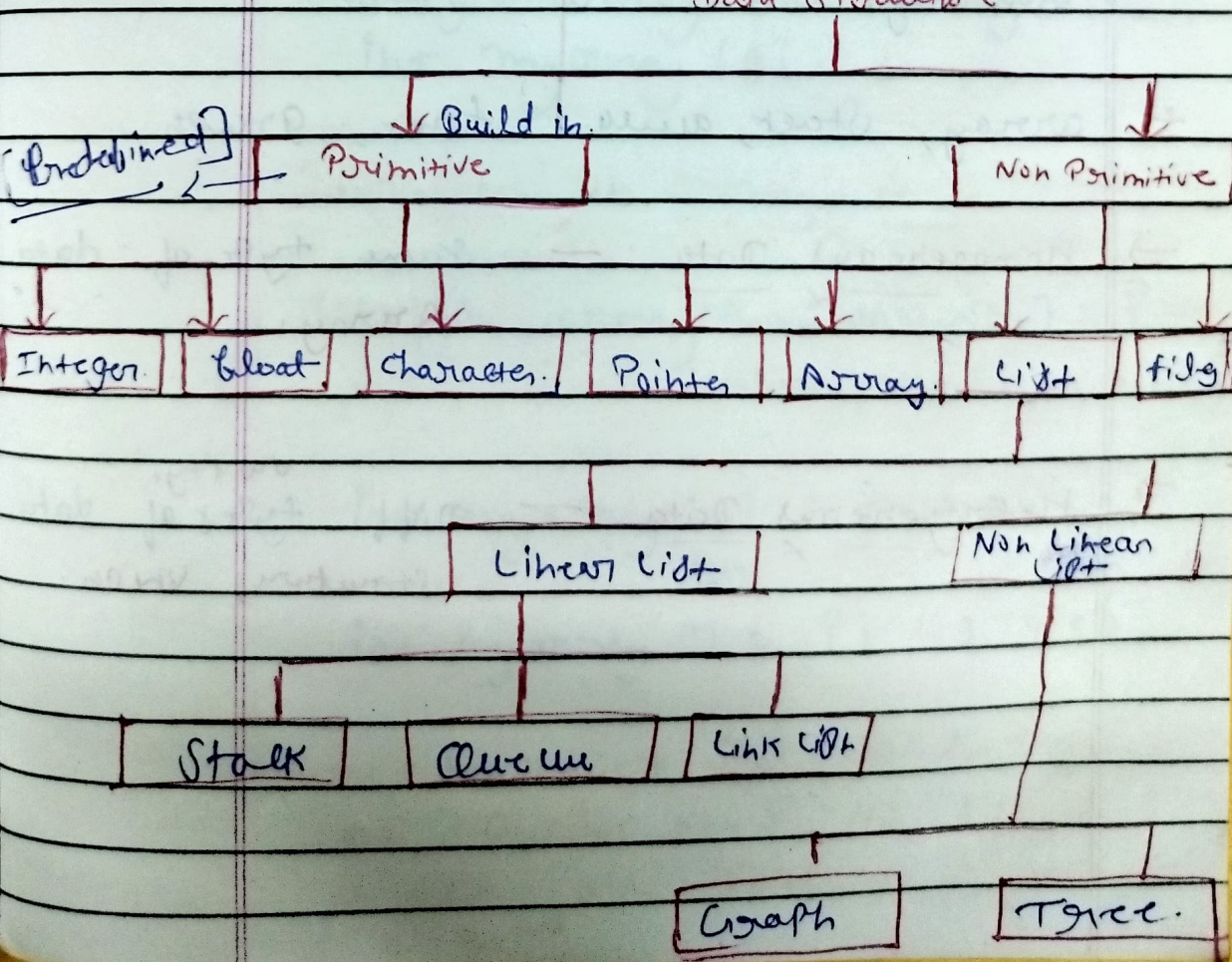
Date:- / /
Page:-

(DSA Theoretical) :—

* Data Structure — Way of organised data in a computer memory (cache, main, secondary) so memory can be used efficiently in both of time / space.

* Organization of data.
* Accessing Method.
* Degree of association. (जितने Node आपस में related हैं)
* Processing Methods (Operation आदि करने)

* Effect of Data Structure :—



Date:- / /
Page:-

Linear Data's	Non Linear Data's
* Data element arrange in liner order. each and every element attached to its previous/next adjacent	non-linear data structure attached in hierarchical manner.
* Single level.	Multiple level.
* implementation easy Comparision to non linear.	Complex.
* Data elements can be traversed in a single run only.	Can't traversed in single run.
* array, Stack, queue.	tree, graph.

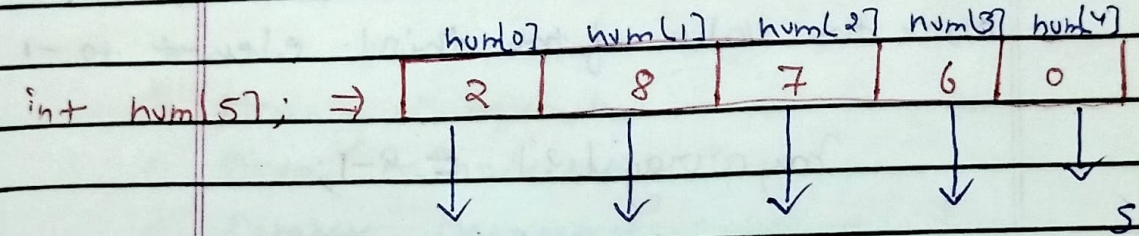
⇒ Homogeneous Data :— Same type of data. Array.

⇒ Heterogeneous Data :— Diff types of data. variety. structure, union.

Collections of homogenious memory location.

Date:- / /
Page:-

* Array :— array store collection of elements of same type stored at contiguous memory locations. and can be accessed using an index.



element1 element2 element3 element4 5

Declare an array in C :—

datatype arrayName [array size];
int myarray [5];

Initialize an array in C :—

datatype arrayName [arraySize] = {
value1, value2, ..., value5};

int myarray [5] = {1, 2, 3, 4, 5};
or
int myarray [] = {1, 2, 3, 4, 5};

Here, we haven't specified the size. the compiler know its size is 5 as we are initializing it with 5 elements.

Date:- / /
Page:-

~~MCQ on PB~~ —

How Java code executed —

Applications of Array —

* Memory Management.
* Data Representation.
* Database Management.
* Implementing Data Structures.
* Caching & Buffering.

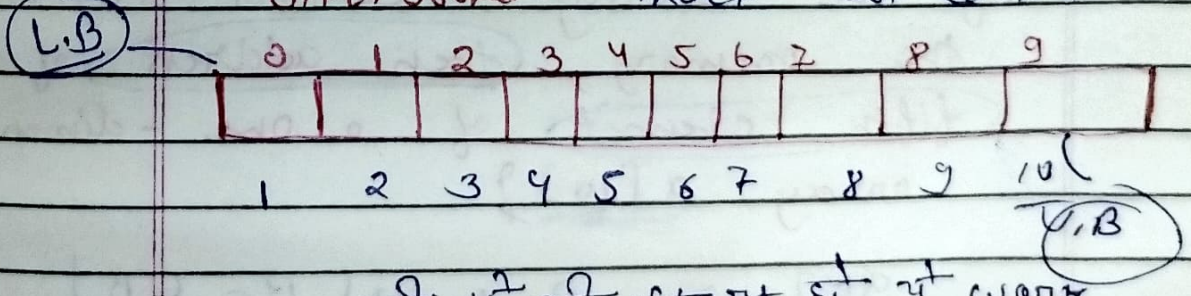
Types of Indexing :—

0 Zero-based indexing — (Default).
1 One - " - " —
n n-based indexing —

Size of an Array :—

Number of element = (Upper bound - Lower bound) + 1
(9 - 0) + 1 = 10

- Lower bound - index first element.
- Upper bound - index last element.



array कहीं से भी start हो पर work करेगा।

4 bits

Date:- / /
Page:-

Size = num of element * Size of each element

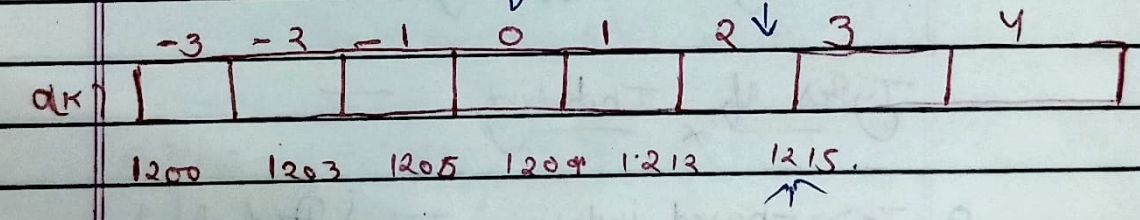
10 × 4B = 40 Bytes

* find Address of the element at k^{th} index

$a[K] = B + w * K$

$a[K] = B + w * (K - \text{Lower bound})$

B - is the base address of the array = 1200
w - Size of each element. = 3
K - index of the element.



$1200 + 3(2 - (-3))$

$1200 + 3(5)$

$1200 + 15 = 1215$

* Let the base address of the first element of the array is 250 and each element of the array occupies 3 bytes in the memory, find address of the fifth element of a one-dimensional array a[10].

$a[K] = B + w(K - LB)$.

Date:- / /
Page:-

in bytes.

MongoDB Compass :— System का server
— x — का connect करने का
लिए।

$$250 + 3[4 - 0)$$
$$250 + 12 = 262$$

An array has been [-6 ... 6] of elements where array element takes 4 bytes, if the base address of the array is 3500 find the address of array [0]?

$$3500 + 4[0 - (-6)]$$
$$3500 + 24 = 3524$$

Date:- / /
Page:-

Two-Dimensional array

2D array is organized as matrices which can be represent as the collection of row/column.

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Initialization of array

data_type array_name [row][column];

int disp[2][4] = {
{10, 11, 12, 13},
{14, 15, 16, 17}
};

or

int disp[2][4] = {10, 11, 12, 13, 14, 15, 16, 17};

R T

D.

Date: / /

* Row major implementation of 2D array

Array are arranged sequentially row by row.

(Default)

1	2	3
4	5	6
7	8	9

Row major →

1	2	3	4	5	6	7	8	9

* Column major implementation

Array are arranged sequentially column by column.

00 (1)	01 (2)	02 (3)
10 (4)	11 (5)	12 (6)
20 (7)	21 (8)	22 (9)

00 (1)	10 (4)	20 (7)	01 (2)	11 (5)	21 (8)	02 (3)	12 (6)	22 (9)

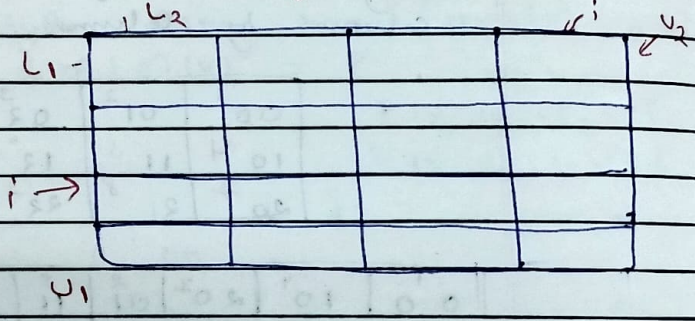
Date:- / /
Page:-

55:55

Row Major implementation of 2D array —

Address of a[i][j] = B + w[(U2 - L2 + 1)(i - L1) + (j - L2)]

B - Base address
w - Size of each element.
L1 - Lower bound of row.
U1 - Upper bound of rows.
L2 - Lower bound of column.
U2 - Upper bound of Column.
(U2 - L2 + 1) = numbers of columns.
(i - L1) = num of row before it.
(j - L2) = num of elem before it.



Row wise -

Address of a[i][j] = B + w*[(U2 - L2 + 1)(i - L1) + (j - L2)]

Column wise -

Addres of a[i][j] = B + w*[(U1 - L1 + 1)(j - L2) + (i - L1)]

Date:- / /

Page:-

3-Dimensional array

A[l1]...[u1], [l2].....[u2], [l3]...[u3]

location of A[i, j, k] =

$$B + (i - l_1)(u_3 - l_3 + 1) + (k - l_3)$$

जो Matrix जिसका ज्यादा से ज्यादा Content 0 होता है।

और जिसमें कम zero होते हैं उसे dense कहते हैं।

Date:- / /
Page:-

Sparse Matrix —

A matrix is large num of its element are zero.

0	0	3	0
0	0	0	8
1	0	3	0
0	0	7	0

* Advantage :—
* Storage Efficiency :—
* Computational Speed :—

Representation :—

* Array representation
* Linked list Representation

→ Array Representation :— 2D array is used to represent a Sparse matrix.

	0	1	2	3	4
0	0	0	3	0	4
1	0	0	5	7	0
2	0	0	0	0	0
3	0	2	6	0	0

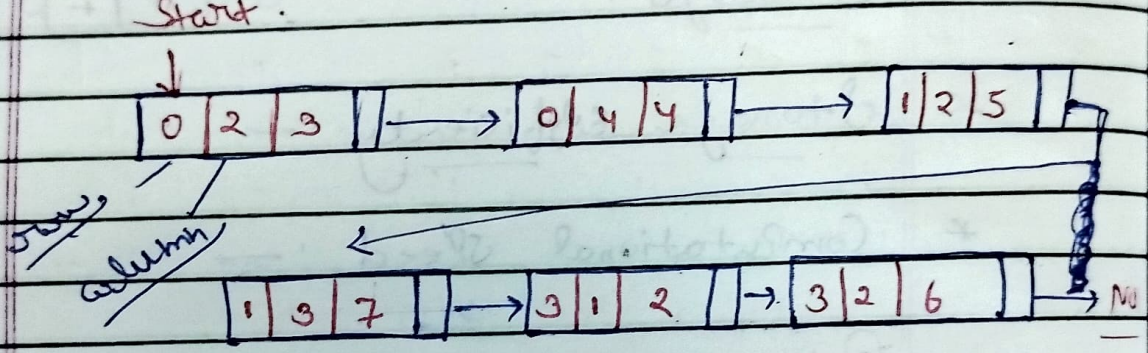
→

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Date:- / /
Page:-

Linked list Representation :—

0	0	3	0	4
0	0	5	7	0
0	0	0	0	9
0	2	6	0	0



Problem with Array :—

* **fixed size / Reallocation :—** have fixed sized, lead to memory waste if the allocated size is larger then the actual data.

* **inefficient insertion / deletion :—** Adding / removing elements in the middles of the array requires shifting the remaining elements, resulting higher time Complexity (O(n)) Compare to linked list.

* **Less flexible :—** can only store elements of the same datatype.

Heterogeneous data type.

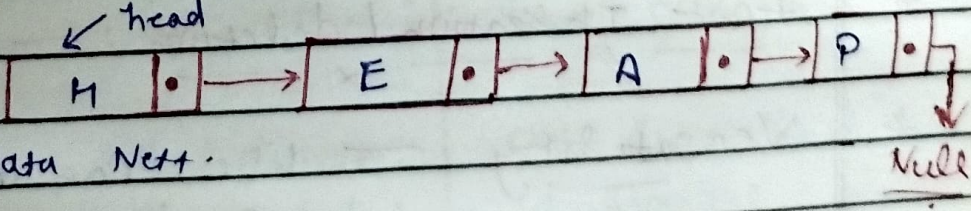
Date:- / /
Page:-

* Linked list ---- * dynamic data structure that consists of elements called nodes, which are connected in a linear sequence.

* Each node contain two part.
  * data.
  * Reference to the next node.

→ first part is the information part of the node, which can store any type of the information. Such as integers, characters, objects.

→ Second part called linked field / next pointer field, contains the address of the next node of the list.



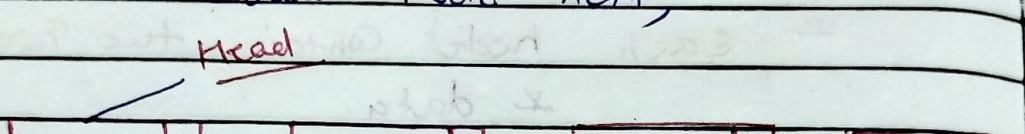
A special case is the list that had no nodes, such a list is called a null list / empty list.

Denoted by - Null Pointer in the variable start / first / head.

Date:- / /
Page:-

Implementation of link list :—

```
Struct node
{
    int data;
    Struct node *next;
};
```



Advantage of link list :—

* **Dynamic size/ efficient memory usage.**

Linked list can easily grow / shrink, allowing for efficient memory allocation.

* **fast Insertion / deletion :—**

* **Versatility :—** We can use various type (singly, doubly, circular) and can store elements of diff data type / objects.

Linked list * Space overhead
* internal/external fragmentation not self.

Date:- / /
Page:-

Disadvantage :—

* Slower access time — have higher time complexity.
Compare to array.

* Memory overhead :— Each node in linked list requires additional memory to store the reference/Pointer to the next node.

Null.

* Pointer manipulation :—

Aspect	Array.	Linked List
Memory Allocation	Contiguous.	Non-contiguous.
Size flexibility	fixed size	Dynamic size, can grow/shrink.
Access time	O(1) direct access due to indexing.	O(n) required traversal.
Insertion/ Deletion.	O(n) worst case shifting may be required.	O(1)
Memory efficiency.	More memory efficient	Extra memory Pointer.

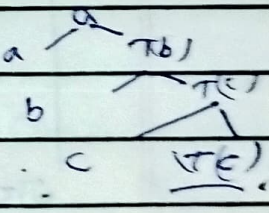
Date:- / /
Page:-

```
#include <stdio.h>
#include <stdlib.h>
// Define Node Structure.

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

Node * CreateNode (int data)
{
    Node * newNode = (Node*) malloc (sizeof (Node));
    if (newNode)
    {
        Printf ("Memory error \n")
        exit (1);
    }
    newNode -> data = data;
    newNode -> next = Null;
    return newNode;
}

int main()
{ Node *head = CreateNode (10);
  head -> next = CreateNode (20);
  head -> next -> next = CreateNode (30);
  Print ("\nLinked List: ");
  traverseList (head);
  return 0;
```

Date:- / /
Page:-

Q → Write a C-Style Pseudocode for traversing a link list iteratively where Pointer head have the address of the first node of the list.

```
Void traverseList (Node *head)
{
    Node * Current = head;
    while (Current != Null)
    {
        Printf ("%d → ", Current → data);
        Current = Current → next;
    }
    Printf ("Null \n");
}
```



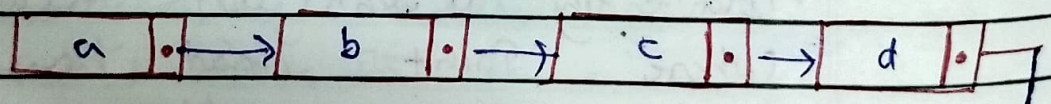
Q → Write a C-Style Pseudocode for traversing a link list recursively, where Pointer head have the address of the first node of the list?

```
Void traverseListRecursive (Node *current)
{
    if (current == Null)
    {
        Printf ("Null \n");
        return 0;
    }
    Printf ("%d → ", current → data);
    traverseListRecursive (current → next);
}
```

Date:- / /
Page:-

Q Write a C-Style Pseudocode for searching a key in a link list iteratively where Pointer head have the address of the first node of this list?

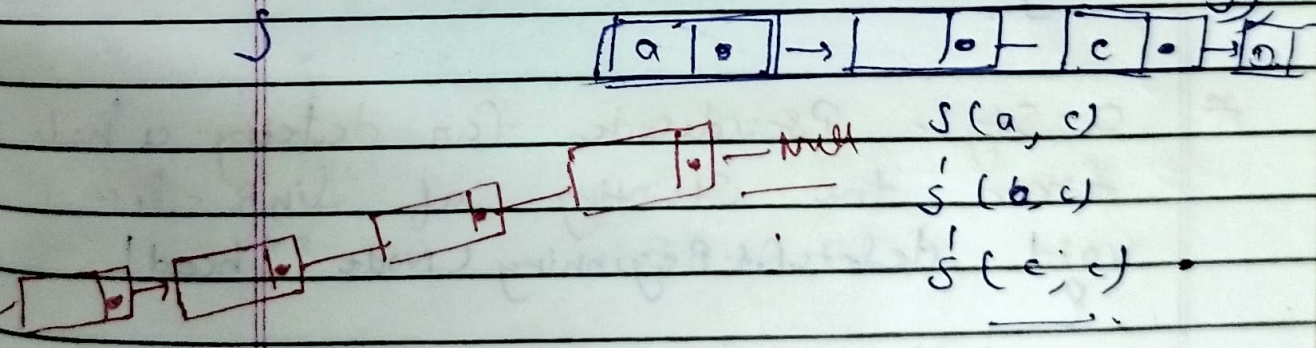
```
Node * SearchKeyIterative(Node *head, int key)
{
    Node * Current = head;
    while (Current != Null)
    {
        if (current -> data == key)
        {
            return current;
        }
        Current = Current -> next;
    }
    return NULL;
}
```



Date: / /
Page:

Q Write a C-Style Pseudocode for Searching a Key in a link list recursively. Where Pointer head have the address of the first node of the list?

```
Node * SearchKeyRecursive (Node *current, int key)
{
    if (current == NULL)
    {
        return NULL;
    }
    if (current -> data == key)
    {
        return current;
    }
    return SearchKeyRecursive (current -> next,
                               key);
}
```



Q C-Style Pseudocode for inserting a node with a key in the starting of Link-list;

```
void insertBeginning (Node **head, int Key)
{
    Node * newNode = CreateNode (Key);
    newNode -> data = data;
    newNode -> next = *head;
    *head = newNode;
}
```

Date:- / /
Page:-

* Write a C-Style Pseudocode for inserting a node with a key after a location in a link-list?

```
Void insertAfter(Node *PrevNode, int key)
{
    if (PrevNode == NULL)
    {
        Printf("the given Previous node
                can't be Null \n");
        return;
    }
    Node *newNode = CreateNode(key);
    newNode -> data = data;
    newNode -> Next = PrevNode -> Next;
    PrevNode -> Next = newNode;
}
```

* C-Style Pseudocode for deleting a node from the starting of link-list.

```
void deleteAtBeginning(Node **head)
{
    if (*head == Null)
    {
        Printf("List is already empty.\n");
        return;
    }
    Node *temp = *head;
    *head = (*head) -> next;
    free(temp);
```

Date:- / /
Page:-

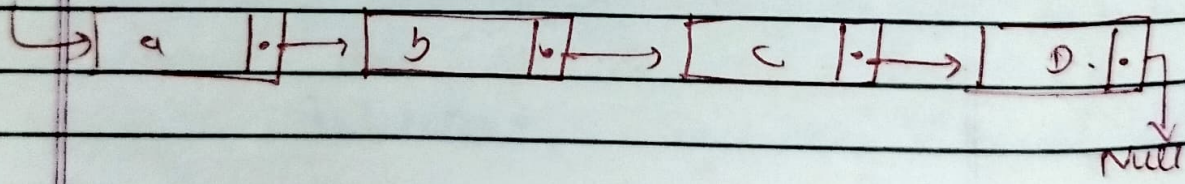
* Write a C-Style Pseudocode for deleting a node after a given location from the starting of link-list.

```
Void deleteAfter (Node *PrevNode)
{
    if (PrevNode == Null || PrevNode->next == Null)
    {
        Printf ("The given node is Null or there's no node after it to delete.\n");
        returning;
    }

    Node *temp = PrevNode->next;
    PrevNode->Next = temp->next;
    free (temp);
}
```

Date:- / /
Page:-

* Write a C-Style Pseudocode for reversing a Link-list in a iterratively?

```
Void reverseList-Iterative (Node **head)
{
    Node * Prev = NULL;
    Node * Current = *head;
    Node * Next = Null;
    While (Current != Null)
    {
        Next = Current -> Next;
        Current -> Next = Prev;
        Prev = Current;
        Current = Next;
    }
    *head = Prev;
}
```



Date: / /
Page:

Write a C-style Pseudocode for reversing a link-list in a recursively

```
Node * reverseListRecursive (Node *head)
{
    if (head == NULL || head -> next == NULL)
    {
        return head;
    }
    Node * rest = reverseListRecursive (
                    head -> next);
    head -> next -> next = head;
    head -> next = NULL;
    return rest;
}
```

Date:- / /
Page:-

The following C function takes a single linked list of integers as a Parameter and rearranges the elements of the list. The function is called with the list containing the integers 1, 2, 3, 4, 5, 6 in the given order. What will be the contents of the list after the function completes execution.

```
Struct node
{
  int value;
  Struct node *next;
};
void rearrange (Struct node *list)
  {
     Struct node *p, *q;
     int temp;
     if ((list) || list -> next)
          return;
     p = list;
     q = list -> next;
     while(q)
     {
        temp = p -> value;
        p -> value = q -> value;
        q -> value = temp;
        p = q -> next;
        q = p? p -> next();
     }
  }
```

1:40

Date:- / /
Page:-

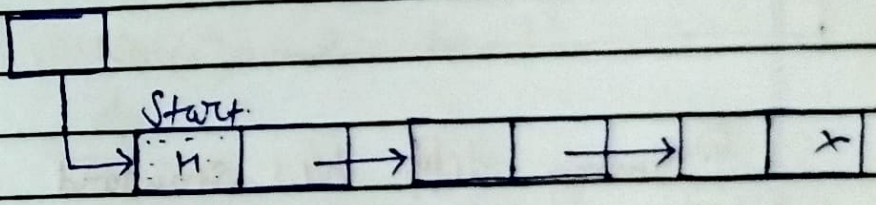
The following function takes a singly-linked list as input argument. It modifies the list by moving the last element to the front of the list and returns the modified list. Some part of the code is left blank. Choose the correct alternative to replace the blank line.

```
typedef struct node
{
    int value;
    struct node *next
} Node;

Node *move_to_front (Node *head)
{
    Node *p, *q;
    if (head == NULL || (head->next == NULL))
        return head;
    q = NULL; p = head;
    while (p->next != NULL)
    {
        q = p;
        p = p->next;
    }
    q->next = NULL; p->next = head; head = p;
    return head;
}
```

Header linked list

* is a variation of standard link list that includes a special node, called the 'header' node, at the beginning of the list.

* header node does't store any actual data; instead, it serves as a reference point that simplifies some operations on the linked list like - inserting, deleting element at the beginning of the list.

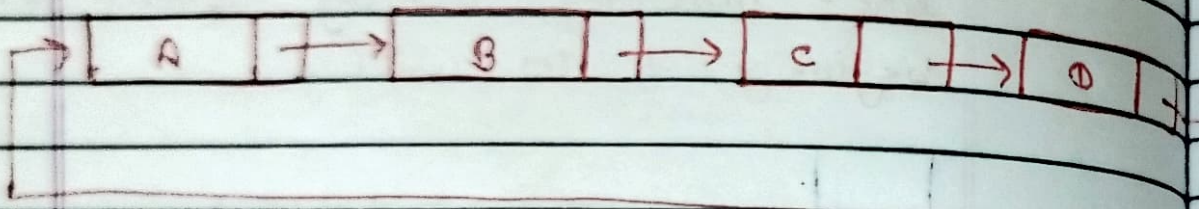



```
Void traverseHeaderLinkedList (Node *header)
{
    if (header == NULL)
    {
        Printf ("List is empty.\n");
        return;
    }
    Node *temp = header -> next;
    While (temp != NULL)
    {
        Printf ("%d -> ", temp -> data);
        temp = temp -> next;
    }
```

printf ("Null\n");
}

Circular Linked List —

* Variation of singly linked list.
* In which the last node in the list point back to the first node.
* Creating loop in circular structure.



Primary diff b/w standard singly linked list and singly circular linked list :-

* in singly circular linked list last node's 'Null' Pointer refer to the first node in the list, rather than being 'NULL'.

* **features** :—

* We used to implement data structures like queues or circular buffers. where element are added to the end and removed from the front. With constant time complexity for both operation.

* Traversal of the list requires a stopping condition. Such as iterating until you reach the starting node again. or use counter to limit the number of iteration. to avoid infinite loops.

```
Void traverseCircularLinkedList (Node *head
{
    if (head == NULL)
    {
        Printf ("List is empty \n");
        return;
    }
    Node * temp = head;
    do
    {
        Printf ("%d →", temp → data);
        temp = temp → next;
    }
    While (temp != head);
    Printf ("%d (head) \n", head → data);
}
```

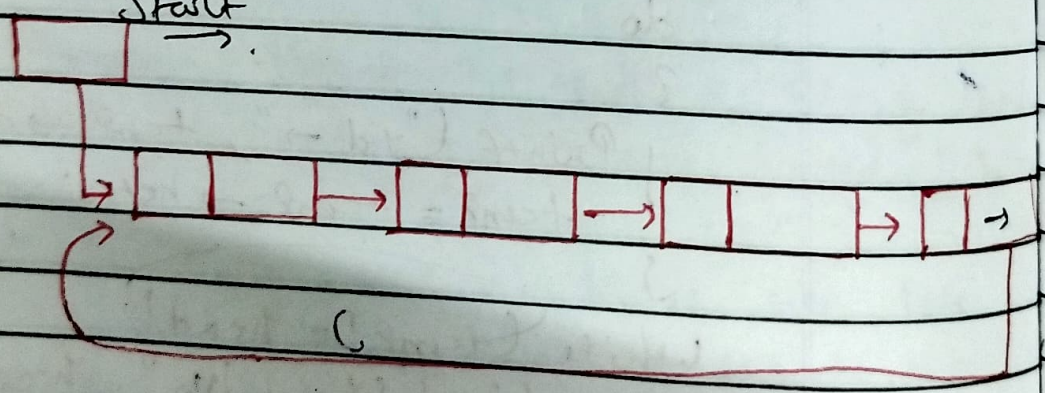
Date:- / /
Page:-

Header Circular link list :—

* Variation of Singly Circular linked list that includes a special node, called the header node, at the beginning of the list.

* The header node does not store any actual data.

* The header node's Primary Purpose is to eliminate the need for special cases when performing certain operations like inserting / deleting elements at the beginning or end of the list.



[Circular header Node]

Date:- / /
Page:-

```
Void traverseHeaderCircularLinkedList(
                        Node *header)
{
  if (header->next == header)
  {
    Printf ("List is empty.\n");
      return;
  }
  Node* temP = header->next;
  While (temP != header)
  {
    Printf( "%d -> " ,temP->data);
      temP = temP->next;
  }
  Printf ("HEADER\n");
}
```

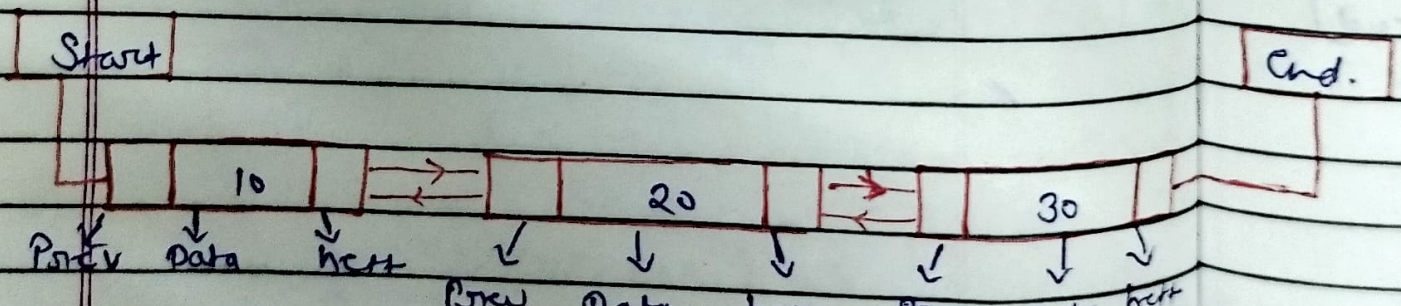
Date:- / /
Page:-

Doubly link list

* In which each node contain data element and two pointers, one pointing to the previous node (previous pointer) and other pointing to the next node (the next pointer) in the sequence.

* The bidirectional linking allows for easier traversal and manipulation of the list in both forward / backward direction.

* as well as simplifying some operations such as insertion and deletion of node at any position in the list.



Date:- / /
Page:-

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node
{
    int data;
    struct Node *Prev;
    struct Node *next;
} Node;

Node* createNode(int data)
{
    Node *newNode = (Node*)malloc(
                    sizeof(Node));
    if (!newNode)
    {
        Printf("Memory error\n");
        exit(1);
    }
    newNode->data = data;
    newNode->Prev = NULL;
    newNode->next = NULL;
    return newNode;
}
Void insertAtBeginning(Node **head, int data)
{
    Node *newNode = CreateNode(data);
    if (*head != NULL)
    {
        (*head)->Prev = newNode;
    }
    newNode->next = *head;
    *head = newNode;
}
```

Pointer)
next' Pointer)

End.

```
Void insertAtEnd (Node **head, int data)
{
    Node * newNode = CreateNode(data);
    if (*head == NULL)
    {
        *head = newNode;
        return;
    }
    Node *temp = *head;
    while (temp -> next != head);
    {
        temp = temp -> next;
    }
    temp -> next = newNode;
    newNode -> Prev = temp;
}

Void deleteAtbeginning (Node **head)
{
    if (*head == NULL)
        return;
    Node *temp = *head;
    *head = (*head) -> next;
    if (*head != NULL)
    {
        (*head) -> Prev = NULL;
    }
    free (temp);
```

Date:- / /
Page:-

```
Void deleteAtEnd (Node **head)
{
    if (*head == NULL)
        return;
    Node *temp = *head;
    while (temp -> next != NULL)
    {
        temp = temp -> next;
    }
    if (temp -> Prev != NULL)
    {
        temp -> Prev -> next = NULL;
    }
    else
    {
        *head = NULL;
    }
    free (temp);
}

Void deleteByPointer (Node **head, Node *loc)
{
    if (*head == NULL || loc == NULL)
        return;
    if (loc -> Prev == NULL)
    {
        *head = loc -> next;
    }
    else {
        loc -> Prev -> next = loc -> next;
        (loc -> next != NULL)
```

Date:- / /
Page:-

```
    {
        loc->next->Prev = loc->Prev;
    }
    free(loc);
}
```

Key features of doubly linked list :

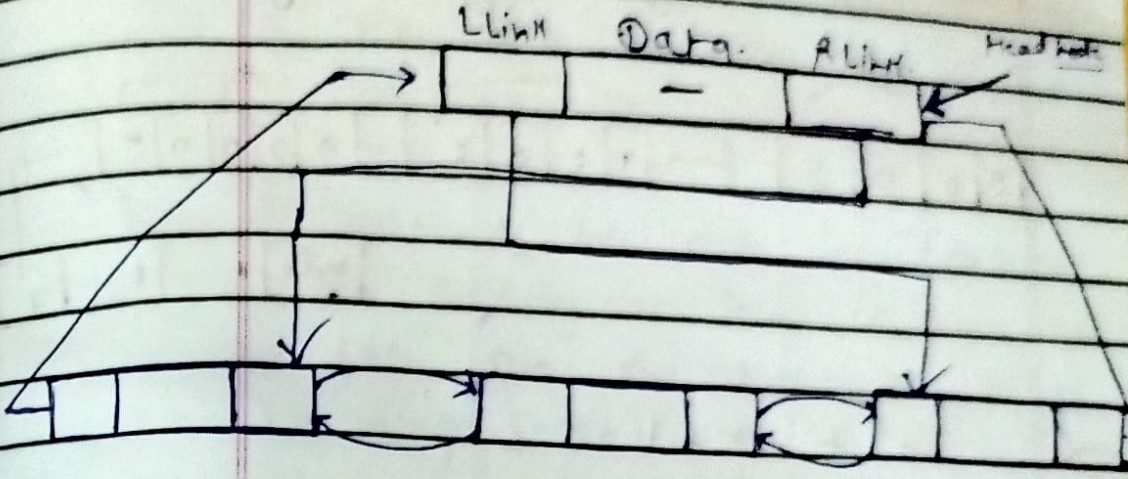
* Each node has two Pointers: 'next' Pointing to the subsequent node and 'Previous' Pointing to the Preceding node in the list.

* first node Previous Pointer and the last node next Pointer are set to the NULL indicating the beginning and end of the list, respectively.

* Doubly linked list allow for easier traversal and manipulation in both forward / backward direction compare to singly linked list.

* Consume more memory than singly linked list due to addition Previous Pointer

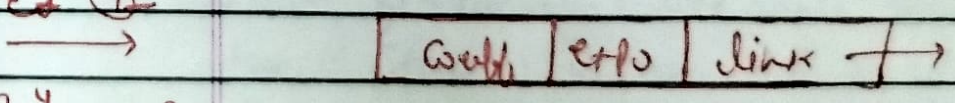
Header Circular Doubly linked list



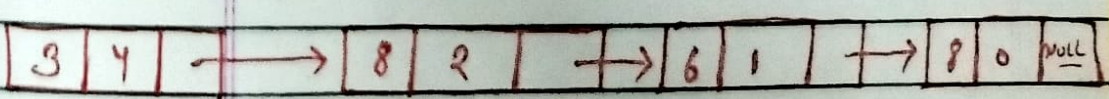
Polynomial Representation using Linked List

* Each node should consist of three elements namely coefficient, exponent and a link to the next term.

* The coefficient field holds the value of the coefficient of a term, the exponent field contains the exponent value of that term and link field contains the address of the next term in the polynomial.



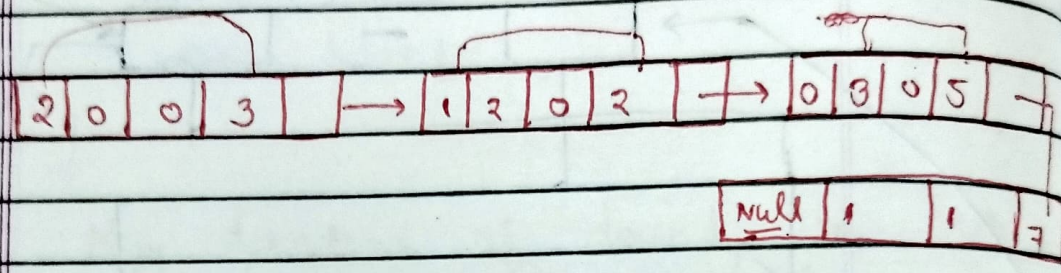
$3x^4 + 8x^2 + 6x + 8$



Date:- / /
Page:-

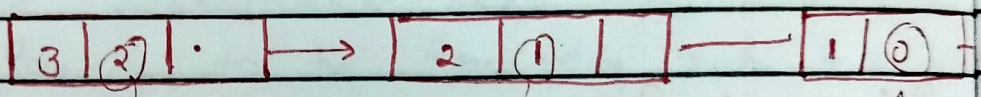
Ex -2

$3x^2 + 2xy^2 + 5y^3 + 7yz$

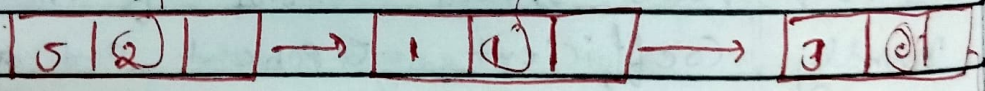


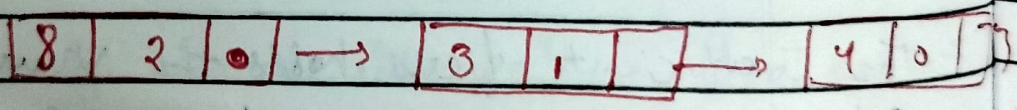
(3)

$3x^2 + 2x + 1$



$5x^2 + 1x + 3$





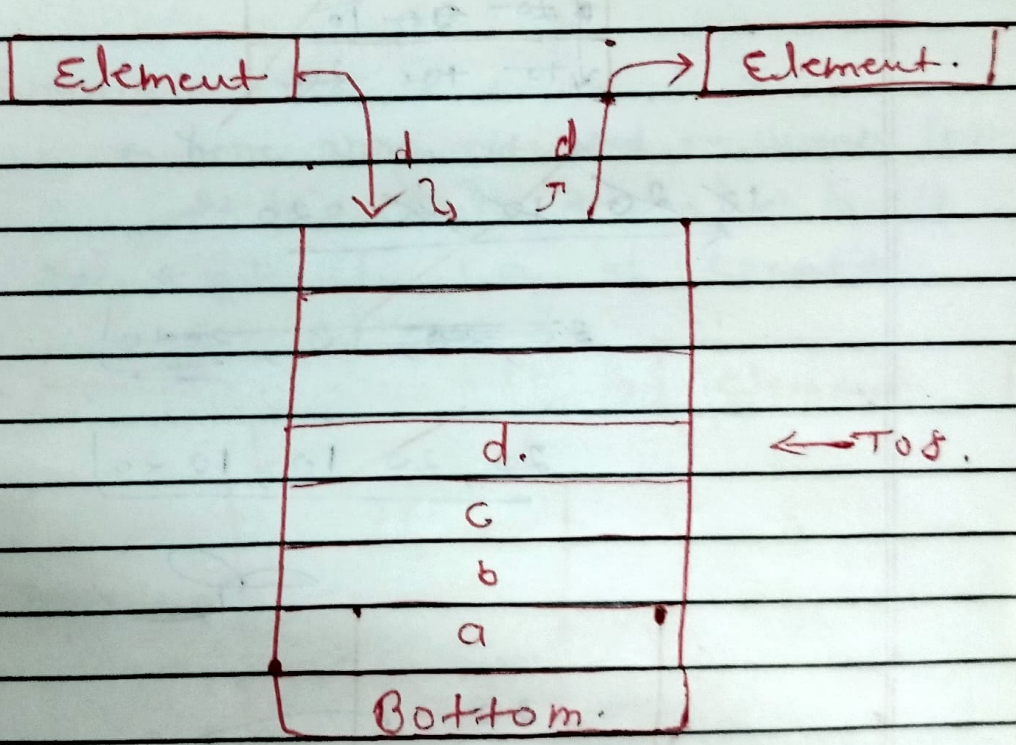
$8x^2 + 3x + 4$

Date:- / /
Page:-

Stack :— [Lifo] / ~~[illegible] fifo~~

A Stack is a non Primitive linear data structure.

* addition of new data item / deletion of existing data item is done from only one end known as top of Stack (TOS)

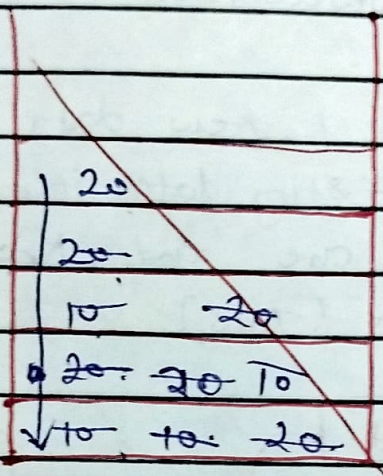


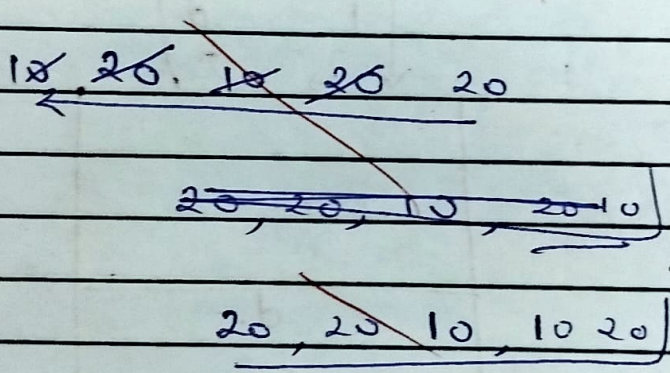
* Most frequently accessible element in the Stack is the top most element; whereas the least accessible element is the bottom of the Stack.

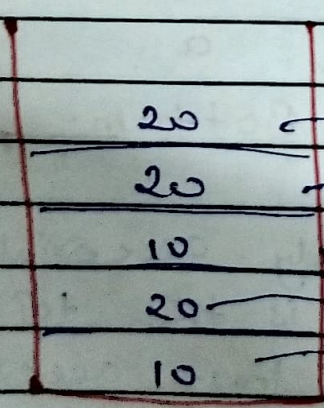
2. 2:4:46

Date:- / /
Page:-

Push(10), Push(20), Pop, Push(10), Push(20), Pop, Pop, Pop, Push(20), Pop. the sequence of values Popped out is.







Push
10, 20, 10, 20

Pop.
20 20 10 10 20

( ) → ( ) → ( )

Date: / /
Page:

Applications of Stack

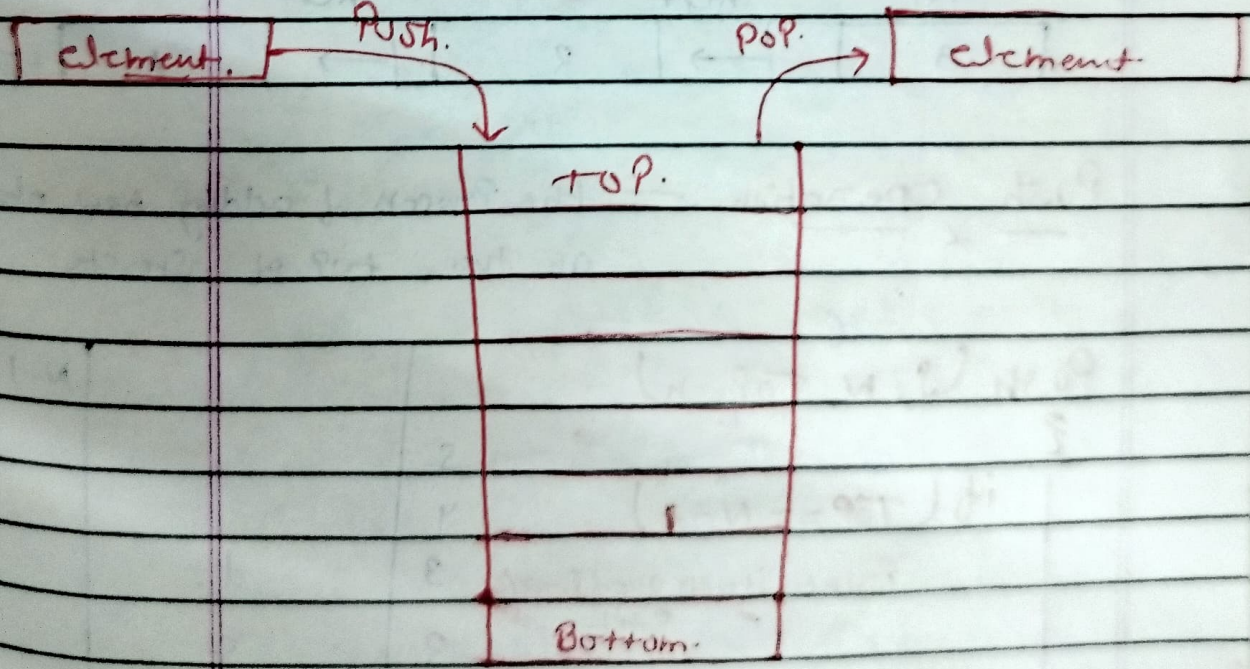
Balanced Parenthesis

* Expression Parsing :- help evaluate/check Programming expression.
* Backtracking :- used in algorithm like - Eight queen Puzzle, maze - solving
* function Call :- memory for detail during call.
* Undo feature :-
* Syntax checking :-

Stack Implementation :- two ways.

* Static Implementation :-

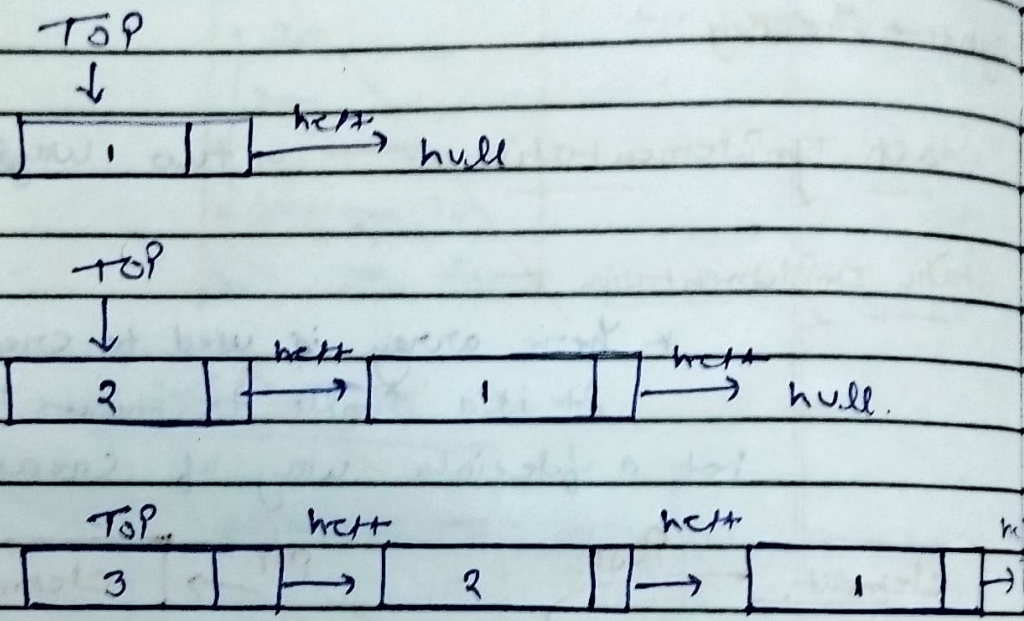
  * here array is used to create stack. It is a simple technique but is not a flexible way of creation.



Date: / /
Page:

Dynamic implementation

It is also called linked list representation and uses Pointer to implement the stack type of data structure.



Push operation :— the process of adding new elements on the top of stack.

```
Push (S, N, TOP, n)
{
    if (TOP == N-1)
        Print Stack overflow / exit.
    TOP = TOP + 1
    Stack[TOP] = n
        exit
}
```

		N-1
5		
4		
3	d.	
2	c	
1	b	
0	a	
-1		

Date:- / /
Page:-

Pop operation — delete elements from top of the stack.

```
Pop (S, N, TOP)
{
    if (TOP == -1)
        Print underflow and exit.
    y = S[TOP]
    TOP = TOP - 1
    return(y) and exit
}
```

4
3
2
1
0
-1

typedef struct

```
{
    int arr[MAX SIZE];
    int top;
} Stack;

void initialize (Stack *S)
{
    S -> top = -1
}

int isEmpty (Stack *S)
{
    return S -> TOP == -1;
}

int isfull (Stack *S)
{
    return S -> TOP == Max_SIZE - 1;
}
```

Null

```
void Push (Stack *S, int item)
{
    if (isfull())
    {
        Printf("Stack is full\n");
        return;
    }
    S->arr[++(S->top)] = item;
}

int Pop (Stack *S)
{
    if (isEmpty(S))
    {
        Printf("Stack is empty\n");
        exit(1);
    }
    return S->arr[(S->top)--];
}

#include<stdio.h>
#include<stdlib.h>
#define MAX_SIZE 100
int main()
{
    Stack S;
    initialize(&S);
    return 0;
}
```

Linked list :-

Date: / /
Page:

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

typedef struct
{
    Node * top;
} Stack;

Void initialize (Stack * S)   // overflow
{
    S -> top = NULL;
}
int isEmpty (Stack * S)
{
    return S -> top == NULL;
}
Void Push (Stack * S, int item)
{
    Node * newNode = (Node *)
    malloc (sizeof (Node));
    if (newNode == NULL)
    {
        Printf ("Stack overflow \n");
        exit (1);
    }
}
```

Date: / /
Page:

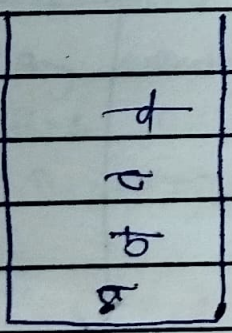
```
newNode -> data = item;
newNode -> next = S -> top;
S -> top = newNode;
}

int Pop (Stack * S)
{
    if (isEmpty(S))
    {
        Printf ("Stack Underflow \n");
        exit(1);
    }
Node * temp = S -> top;
int PoppedData = temp -> data;
S -> top = S -> top -> next;
free (temp);
return (PoppedData);
}

#include <stdio.h>
#include <stdlib.h>
int main()
{
    Stack S;
    initialize (&S);

    return 0;
}
```

Date:- / /
Page:-

```
Void reverseString (char str[])
{
    int length = strlen(str);
    Stack S;
    initialize(&S);
    for(int i=0; i<length; i++)
    {
        Push(&S, str[i]);
    }
    for(int i=0; i<length; i++)
    {
        str[i] = Pop(&S);
    }
}
```

Push.

a b c d



Pop.

d, c, b, a.

```
int main()
{
    char str[] = "Hello, World";
    Printf("original String: %s\n", str);
    reverseString(str);
    Printf("Reverse String: %s\n", str);
    return 0;
}
```

2:22:22

Date:- / /
Page:-

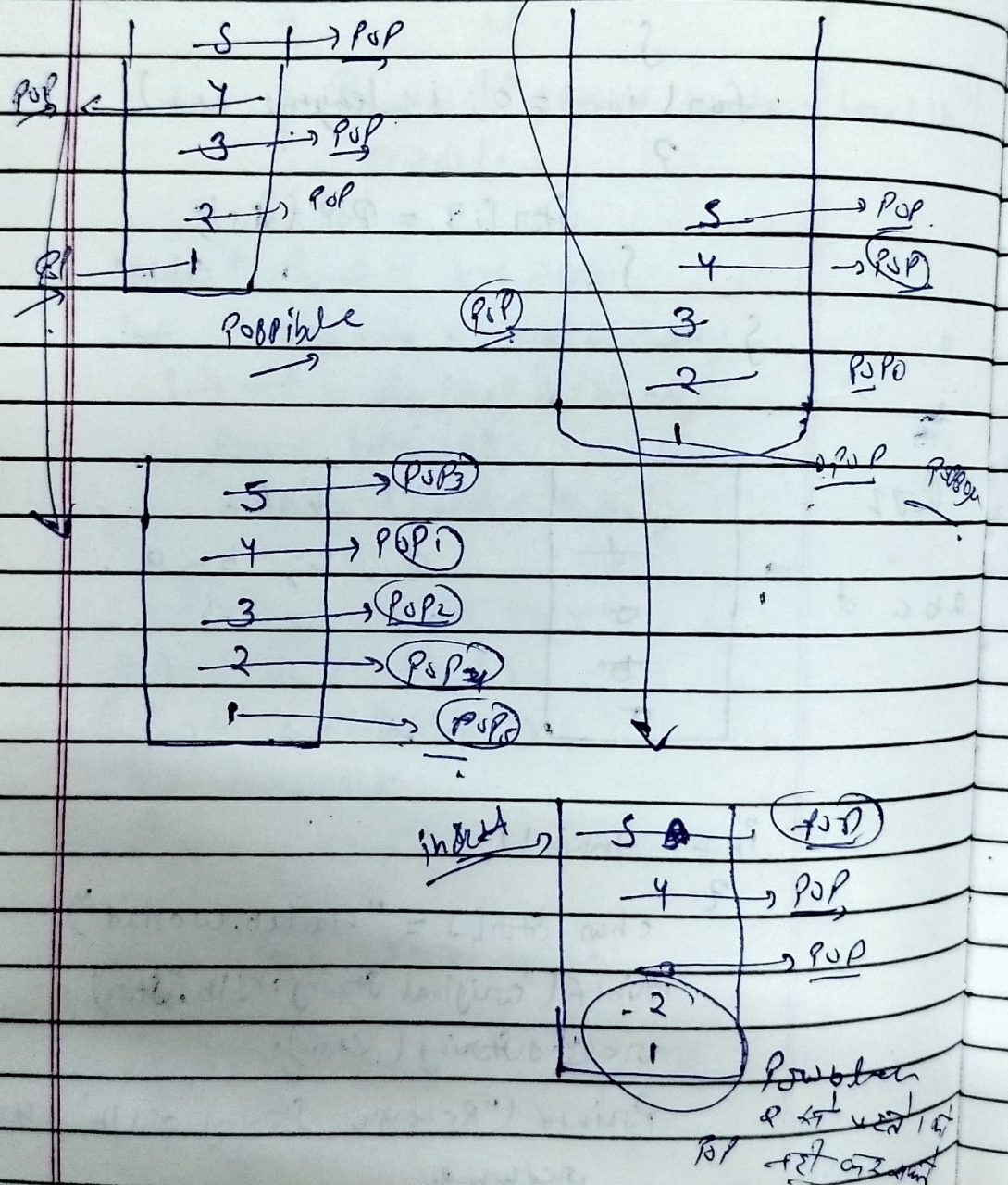
* if the input sequence is 1, 2, 3, 4, 5 then identify the wrong stack Permutation. Possible (POP sequence).

3, 5, 4, 2, 1

2, 4, 3, 5, 1

4, 3, 5, 2, 1

4, 3, 5, 1, 2



Date:- / /
Page:-

Notation :-

Infix notation :- Operator is written in b/w the operands E.g A+B.

Prefix notation :- Operator is written before the operands. also called Polish notation
+ AB.

Postfix :- Operator are written after the operands. also called : Suffix / reverse Polish notation.
AB+

* Postfix notation is suitable for a Calculation.
* Any expression entered into the computer is first converted into Postfix notation stored in Stack and then calculate.

Eg → a + b * c / d ^ e ^ f * d - c Convert Prefix Postfix.

a + b * c / d ^ e f ^ ^ * d - c

a + b * c / d e f ^ ^ ^ * d - c

a b c * / d e f ^ ^ ^ * + d - c

^, /, *, +, -

Date:- / /
Page:-

a + b * c / d ^ e ^ f * d - c Convert it into both Prefix / Post fix.

Post fix →

a + b * c / d ^ e ^ f * d - c

a + b * c / d ^ e f ^ * d - c

a + b * c / d e f ^ ^ * d - c

a + bc * / def ^ ^ * d - c

a bc * def ^ ^ / d * - c

~~a bc * def ^ ^ / * d * - c~~

~~a bc * def ^ ^ / * d dc -~~

a + bc * def ^ ^ / d * + - c

a bc * def ^ ^ / d * + c -

Prefix →

Left →

a + b * c / d ^ e ^ f * d - c

a + b * c / d ^ ^ e f * d - c

a + b * c / ^ d ^ e f * d - c

a + * bc / ^ d ^ e f * d - c

a + / * bc ^ d ^ e f * d - c

a + * / * bc ^ d ^ e f d - c

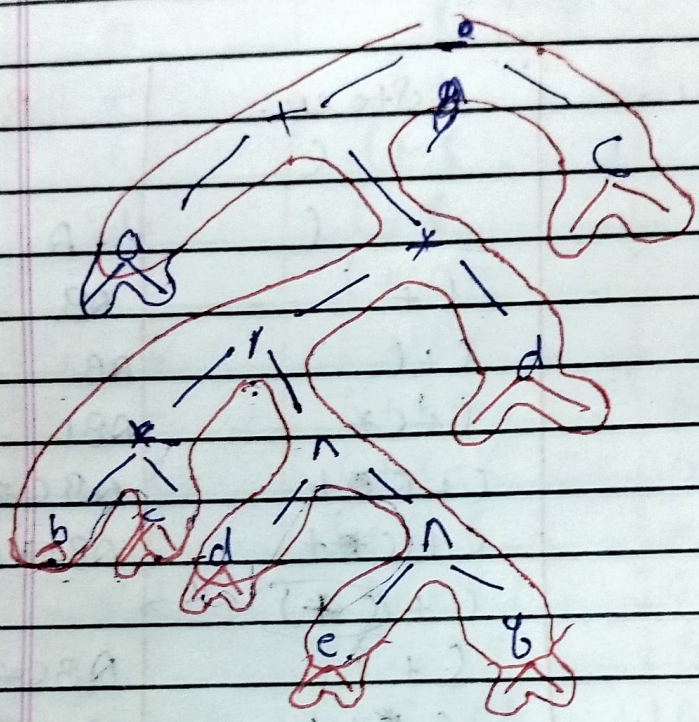
+ a * / * bc ^ d ^ e f d - c

- + a * * bc ^ d ^ e f d c

Date:- / /
Page:-

Tree of :-

Left → a + b * c / d ^ e ^ f * d - c

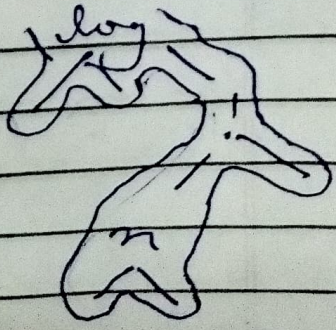


Prefix :- - + a * / * b c ^ d ^ e f d c

infix - a + b * c / d ^ e ^ f * d - c

Postfix - a b c * d e f ^ ^ d * + c -

① Consider an expression log(n!). Convert it into both Prefix and Post fix notation. log ! n.



Prefix - log ! n

infix log n !

Postfix ~~log~~ n ! log

Less से high हो सकता है
+ *
(—)

but high से Less
* + नहीं

Date: / /
Page:-

Stack से :—

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

Character	Stack	Postfix
(	(	
A	(	A
+	(+	AB
(	(+(	AB
*	(+(*	ABC
+	(+(+	ABC*
D	(+(+	ABC*D
)	(+(*)	
	(+	ABC*D+
/	(+/	ABC*D+
E	(+/	ABC*D+E
)	(	ABC+D+E/+

(+
higher

Date:- / /
Page:-

A + (B * C - (D / E ^ f) * H)

Character	Stack	Postfix
A	([initial]	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
(	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	ABC*D
E	(+(-(/	ABC*DE
^	(+(-(/^	ABC*DE
f	(+(-(/^	ABC*DEf
)	(+(-	ABC*DEf^/
*	(+(-*	ABC*DEf^/
H	(+(-*	ABC*DEf^/H
)	(+	ABC*DEf^/H*-
		ABC*DEf^/H*-+

Evaluation of Arithmatic operation :—

↳ 10 * 5 + 100 / 100 - 5 ÷ 7 % 3.

50 + 10 - 5 ÷ 7 % 3.

50 + 10 - 5 ÷ 1

60 - 5 ÷ 1

55 ÷ 1

55

① M M M +

2:50:42

Date:- / /
Page:-

Evaluate the Post-fix expression —

8 2 3 * 1 1 + 4 1 * 3 1 +

14 + 2 = 16

2	
3	4 1 2
4	
1	4 * 1
4	
14	8 + 6 = 14
6	
1	6 1 1 = 6
6	
3	
2	2 * 3 = 6
8	

Evaluate the Prefix expression :—

+ + . 8 1 * 2 3 1 1 * 4 1 2.

A → L

14	14 + 2 = 16
8	
6	6 + 8 = 14
6	
2	6 / 1 = 6
3	2 × 3 = 6
1	
2	4 / 2 = 2
4	
4	4 * 1 = 4
1	
2	

Date:- / /
Page:-

Algo
① Add right Parenthesis ")" to P.
[This act as a Sentinel]

② Scan P from left to right and repeat Step 3 and 4 for each element of P until the Sentinel ")" is encountered.

③ if an operand is encountered, Put it on Stack.

④ if an operator ⊛ is encountered then.
(a) Remove the top two element of Stack where A is the top element B is the next to top element.

(b) Evaluate B ⊛ A.

c Place the result of (b) back on STACK
[End of if structure]
[End of Step 2 loop]

⑤ Set value equal to top element on STACK.

⑥ END.

Date:- / /
Page:-

Recursion → When function call itself.

f(n) = f(n-1) + 1 → f(n-2) + 1 →

* Base Case :— Provides a direct solution without further recursive call.

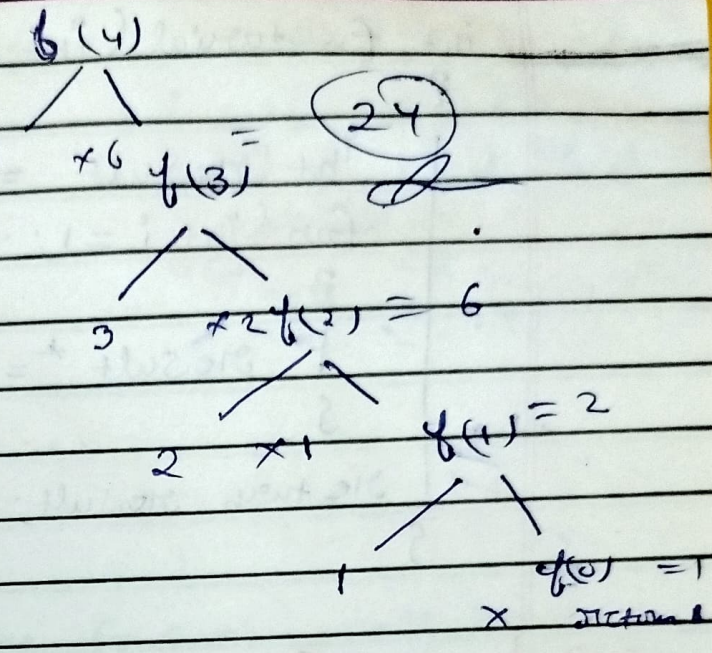
* Recursive Case :— fun calls itself to address smaller instances of Problem.

* Call Stack :— एक fun ने दूसरे fun को call किया दूसरे ने 3 को ना एक chain को याद करना call Stack. होता है।

Deep recursion cause a Stack overflow error.

```
int factorial(int n)
{
    if(n==0)
    {
        return 1; // Base Case : factorial of 0 is 1
    }
    else
    {
        return n * factorial(n-1) // Recursive Case.
    }
```

Date:- / /
Page:-



Iteration :—

→ * Process of Repeatedly executing a set of statement, All along as specific condition remains true.

Loop type :—

for loop : used for known num of repetition.
while loop : Runs as long as condition is true
Do-while : execute at least one before checking the condition.

Control Statements :—

- Break : Exit the loop.
- Continue : Skip to the next iteration.

Nested loop : — Loops within loops

Date: / /
Page:

```
int factorial (int n)
{
    int result = 1;
    for (int i = 1; i <= n; i++)
    {
        result *= i;
    }
    return result;
}
```

Aspect	Recursion	iteration
Basic Concept	function calls itself to solve sub Problem.	Use loop to repeatedly execute code block.
Memory Usage	Uses more memory.	Uses less memory
Termination →	requires base case to prevent infinite loop.	Requires a loop exit condition
Performance	Might be slower due to overhead of fun call.	typically faster due to direct loop machinics.

Date:- / /

Page:-

① Direct Recursion — A function calles itself directly.

* Tail Recursion — पहले Print call बाद में

* Head Recursion — पहले call और बाद का काम फिर Pending काम।

* Indirect Recursion —

Two or more funcation call each other in a cyclic manner.

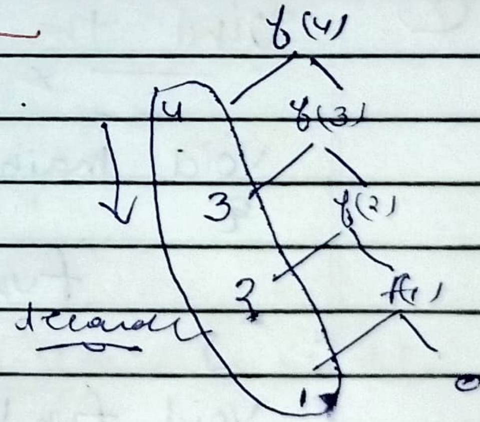
for ex :- fun A call fun B, and fun B calls fun A.

find the O/P of —

(Tail Recursion)

```
void main()
{
    fun(4);
}
void fun(int x)
{
    if (x>0)
    {
        printf("%d", x);
        fun(x-1);
    }
}
```

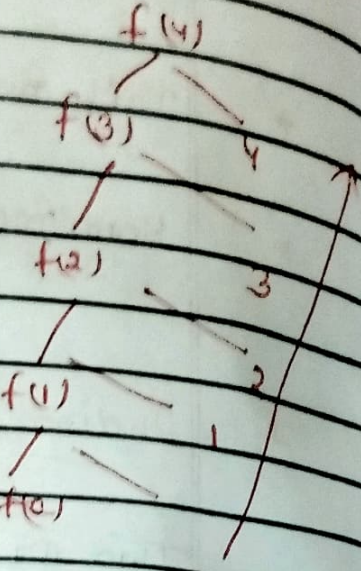
call बाद में →



Date / /
Page

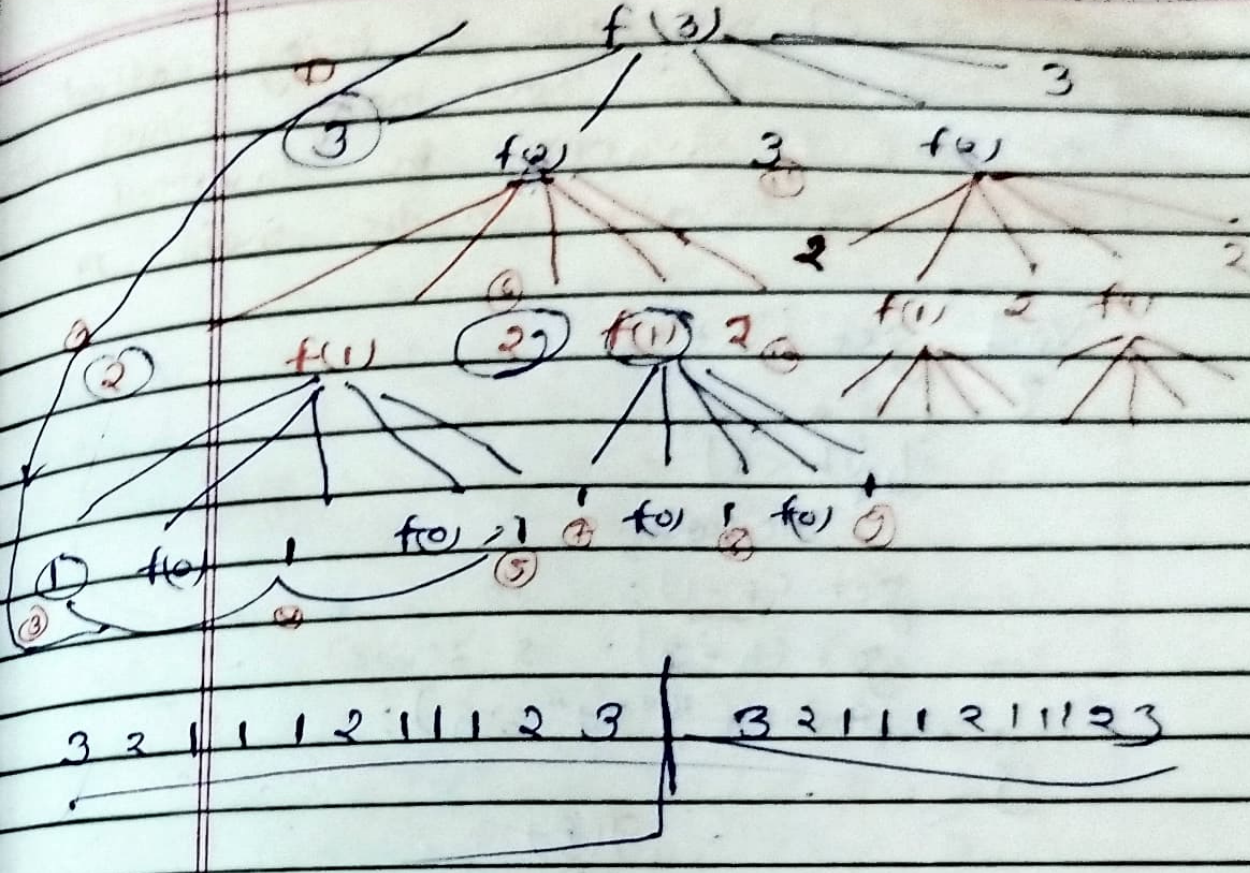
Head Recursion

```
Void main()
{
    fun(4);
}
Void fun(int n)
{
    if(n>0)
    {
        fun(n-1);
        Print("%d", n);
    }
}
```



(X) Find the output :—

```
Void main()
{
    fun(3);
}
Void fun(int n)
{
    if(n>0)
    {
        Printf("%d", n);
        fun(n-1);
        Printf("%d", n);
        fun(n-1);
        Printf("%d", n);
    }
}
```



(*) Find the output of following code n = 5;

```
int x(int n)
{
    if(n<3)
        return 1;
    else
        return x(n-1) + x(n-1) + 1;
}
```

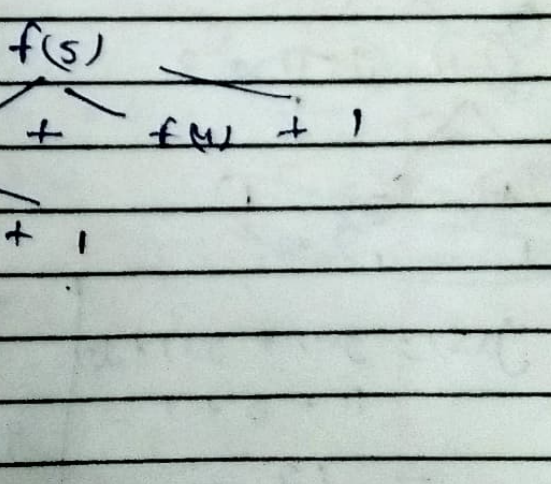
f(3) = 3

f(4) = f(3) + f(3) + 1 = 7

f(5) = f(4) + f(4) + 1

= 7 + 7 + 1

f(5) = 15



3 : 16 : 17

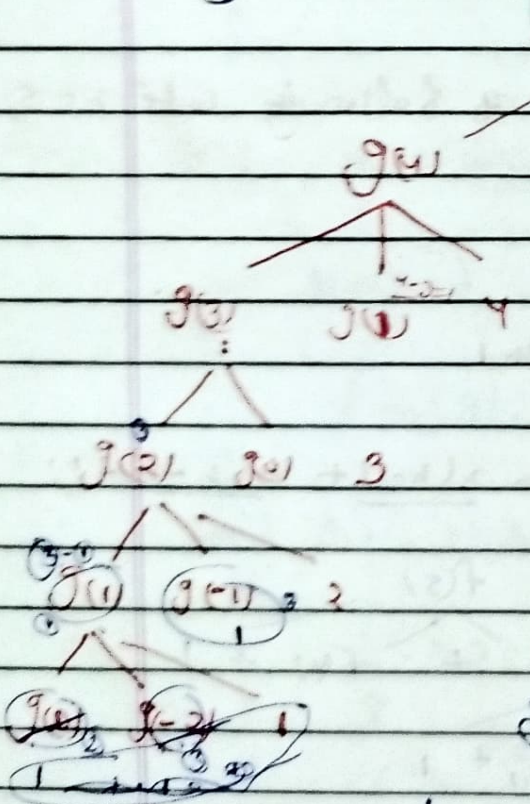
call find a, s ?

Date: / /

Page:

Q. if get(5) function is being called in main() then how many times the get() function be invoked before returning to the main()?

```
void get (int n)
{
    if (n < 1)
        return;
    get (n-1);      5-1
    get (n-3);      5-3 = (2)
    printf ("%d", n);
}
```



calls find ?

$g(-2) = 1$

$g(-1) = 1$

$g(0) = 1$

$g(1) = 3$

~~$g(2) = g(1) + 1 + 1 + 3 = 6$~~

$g(2) = g(1) + g(-1) + g(2)$
$3 + 1 + 1 = 5$

$g(3) = g(2) + g(0) + g(3)$
$5 + 1 + 1 = 7$

$g(4) = g(3) + g(1) + g(4)$
$7 + 3 + 1 = 11$

$g(5) = g(4) + g(2) + g(5)$
$11 + 5 + 1$
$= 17$

Date:- / /
Page:-

⊗ The fibonacci num commonly denoted f_n, form a sequence, called the fibonacci sequence, such that each num is the sum of the two preceding ones, starting from 0 to 1.

$f(0)=0$ if $n==0$, then $f(n)=0$.

$f(1)=1$ if $n==1$, then $f(n)=1$

if $n>1$, then $f(n-1)+f(n-2)$.

$f(n) = f(n-1) + f(n-2)$

$f(2) = f(2-1) + f(2-2)$

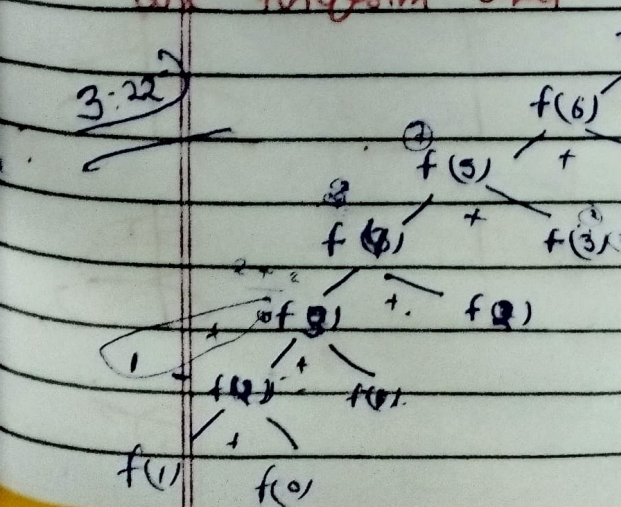
$f(1) + f(0)$

$f(2) = 1 + 0 = 1$

$f(3) = f(2) + f(1) = 2$

n	0	1	2	3	4	5	6	7	8	9	10	11
f(n)	0	1	1	2	3	5	8	13	21	34		
No of invocation	1	1	3	5	9	17	25	41				
No of addition	0	0	1	2	4	7	12	20				

it is easy but ask in f(7) how many addition are performed and how many times f() called



3.22

34-1=33

f(9) - 1

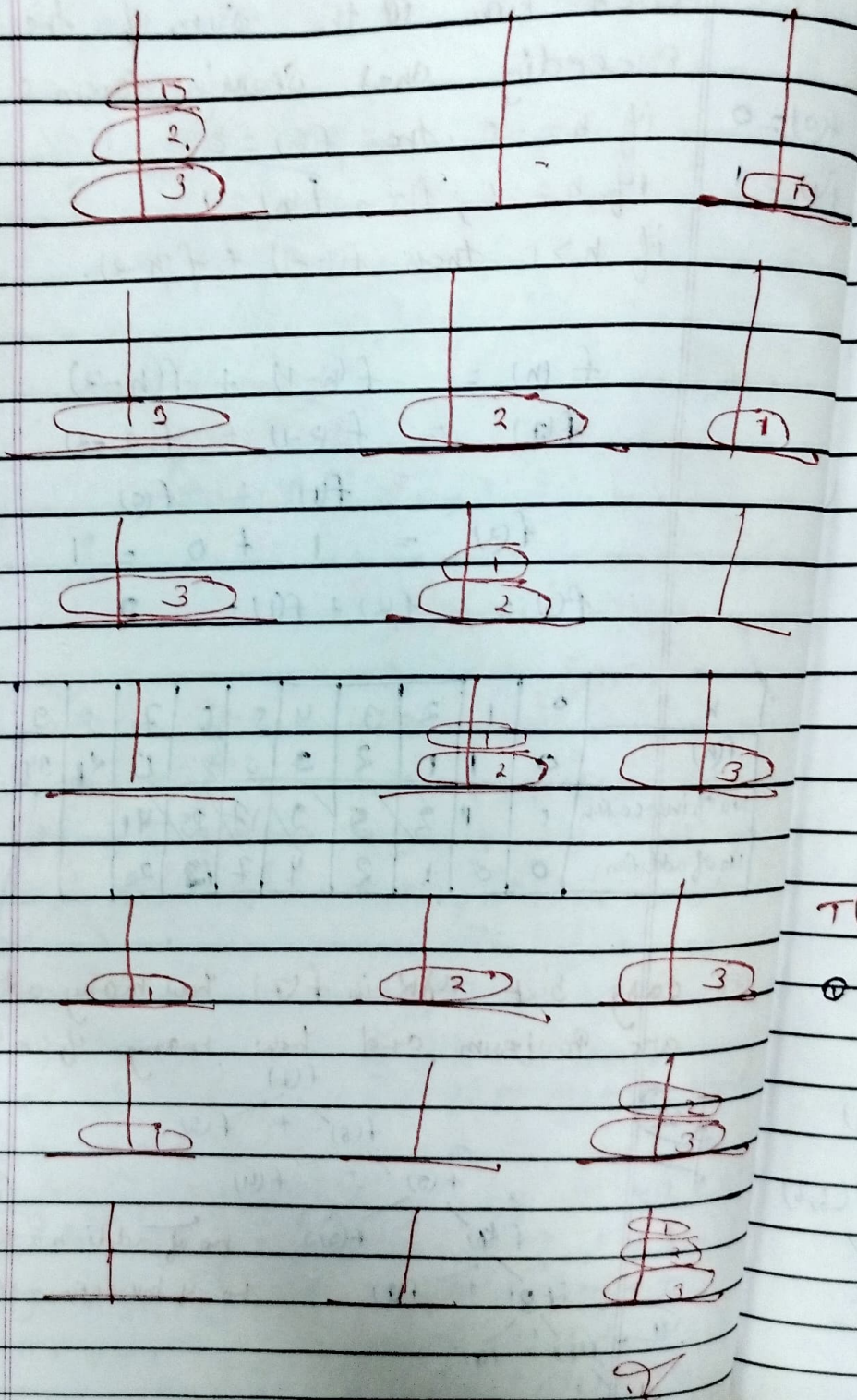
no of addition = f(n+1) - 1

no of invocation = 2f(n+1) - 1

Date:- / /

Page:-

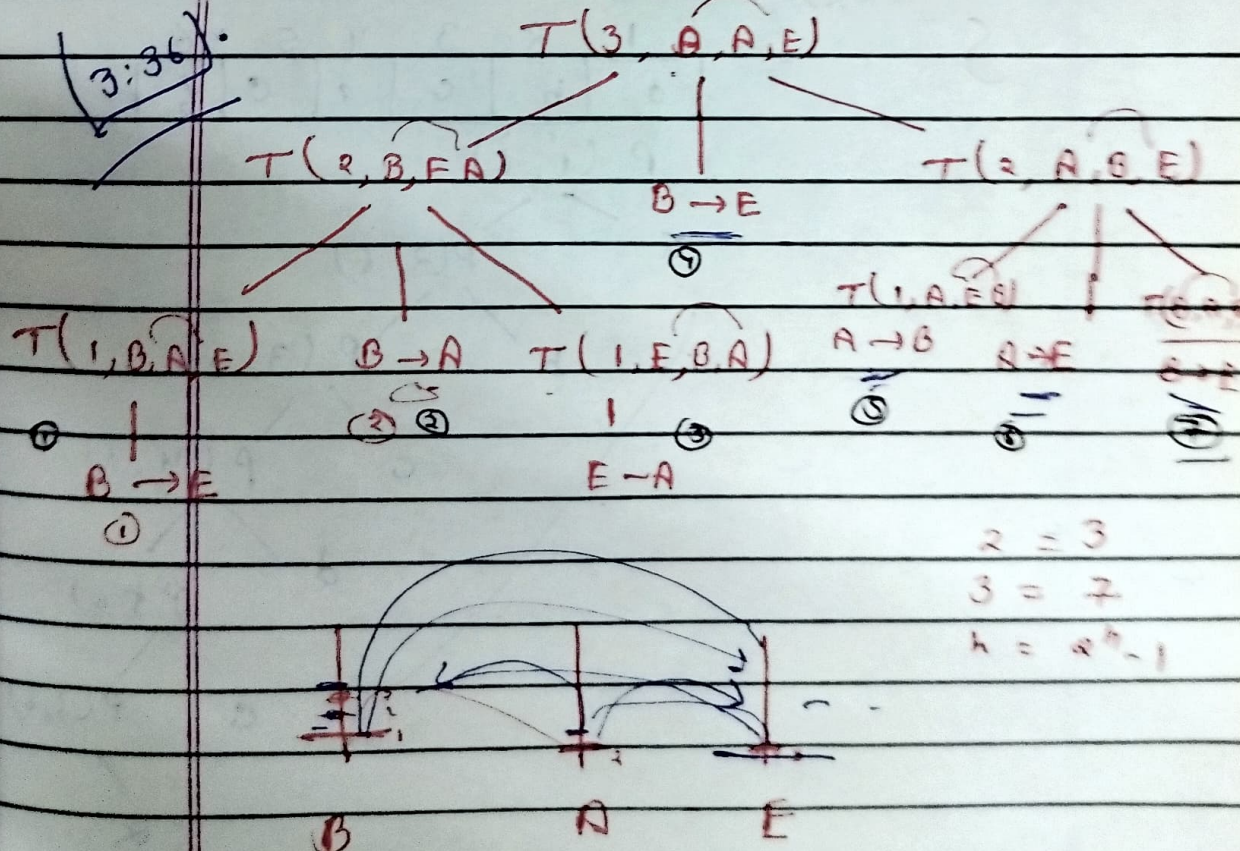
Tower of Hanoi :— / Brahma :—



Date:- / /
Page:-

Tower (N, B, A, E)
{
if (h = 1)
{
B → E
return;
}
tower (h-1, B, E, A);
B → E
tower (h-1, A, B, E)
Return;

total disk moves = $2^h - 1$
" num of fun call = $2^{h+1} - 1$
How many invocation are occurring for the first disk to move = h.



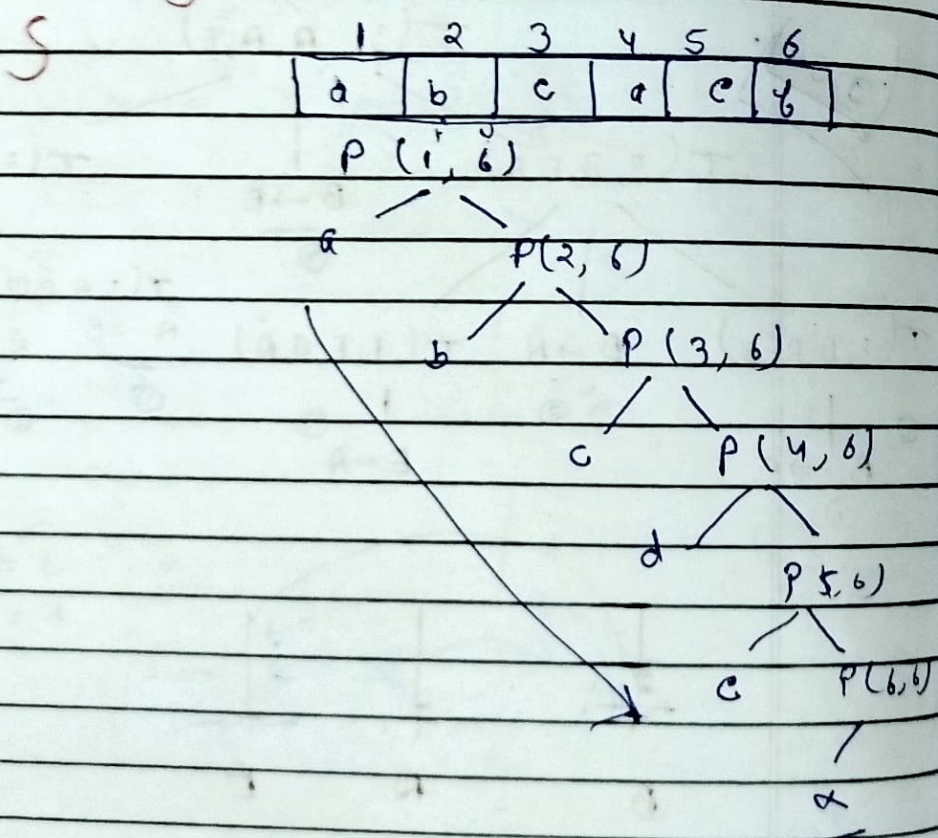
2 = 3
3 = 7
h = $a^n - 1$

Date: / /
Page:

Init Recap

Q: find the o/p —

```
void Print_array (a, i, j)
{
    if (i == j)
    {
        Printf ("%d", a[i]);
        return;
    }
    else
    {
        Printf ("%d", a[i]);
        PrintfArray (a, i+1, j)
    }
}
```

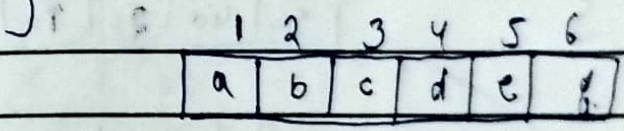


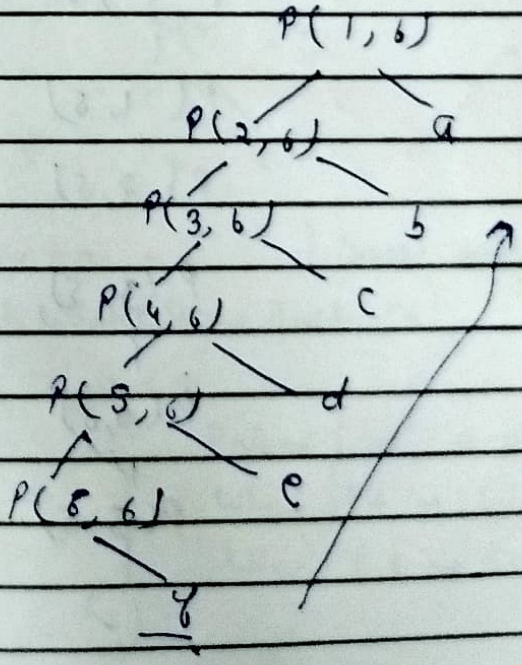
Date: / /
Page:

Head →

find the op ——

```
Void Print_array (a, i, j)
{
    if(i==j)
    {
        Printf("%d", a[i]);
        return;
    }
    else
    {
        Print_array(a, i+1, j)
        Printf("%d", a[i]);
    }
}
```

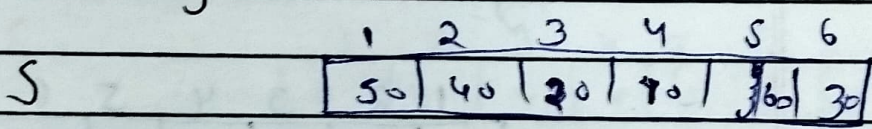




Date:- / /
Page:-

Q Void Print_Somthing (a, i, j)

```
{
    if (i == j)
    {
        Printf ("%d", a[i]);
        return;
    }
else
{
    if (a[i] < a[j])
        Print_Somthing (a, i+1, j);
    Else
        Print_Somthing (a, i, j-1);
}
```



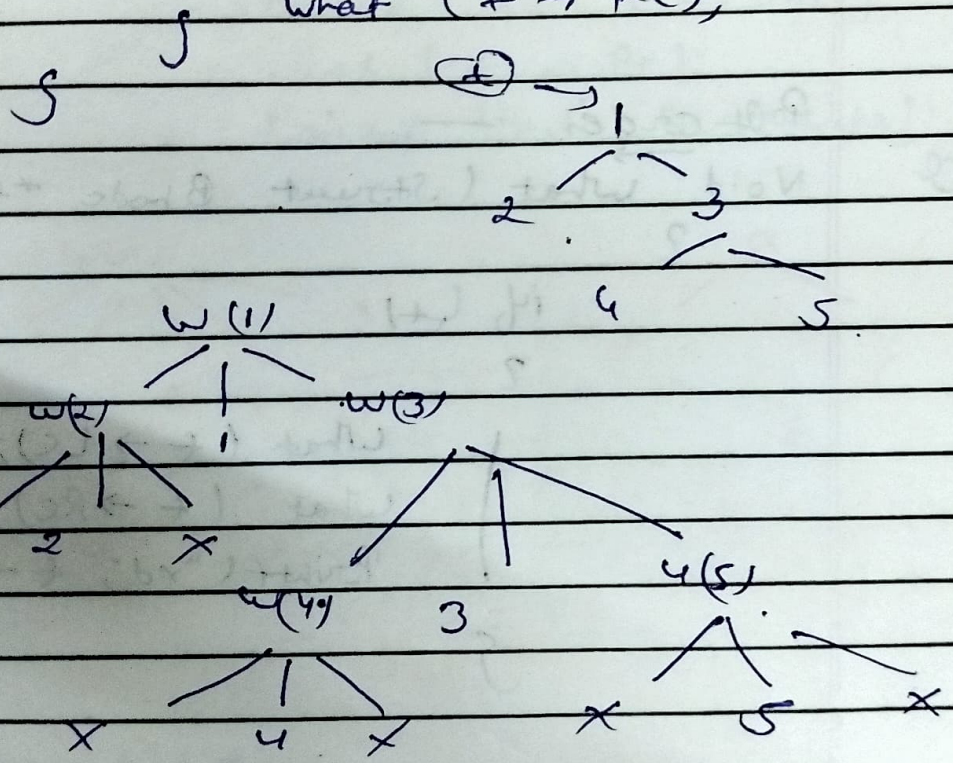
P (1, 6)
|
P (1, 5)
|
P (2, 5)
|
P (3, 5)
|
P (4, 5)
|
P (5, 5)

)

Date:- / /
Page:-

what this function is doing :—

```
Void What (Struct Node *t)
{
    if (t)
    {
        What (t -> LC);
        Printf ("%d", t -> data);
        What (t -> RC);
    }
}
```



(2) Preorder — [inverse traversal].

```
Void What (Struct Node *t)
{
    if (t)
    {
        Printf ("%d", t -> data);
        What (t -> LC);
        What (t -> RC);
    }
}
```

LC
RC

Date: / /
Page:

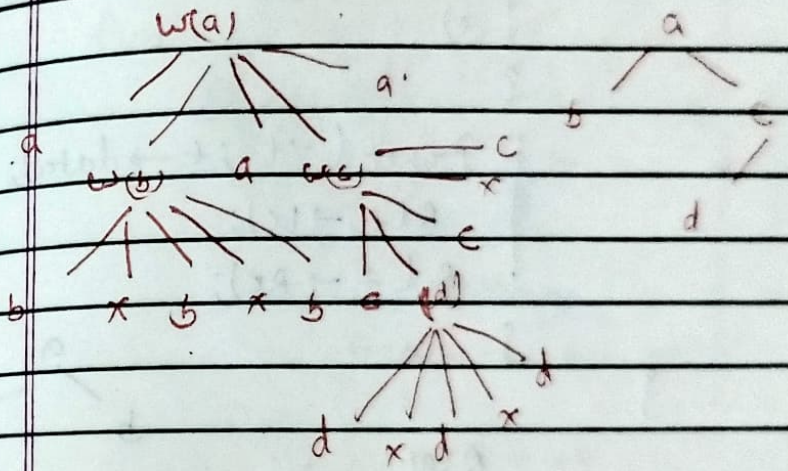
Postorder :—

Q
```
Void what (struct Bnode *t)
{
    if (t)
    {
        what (t->LC);
        what (t->RC);
        Printf("%d", t->data);
    }
}
```

Date: / /
Page:

Find what function doing ---

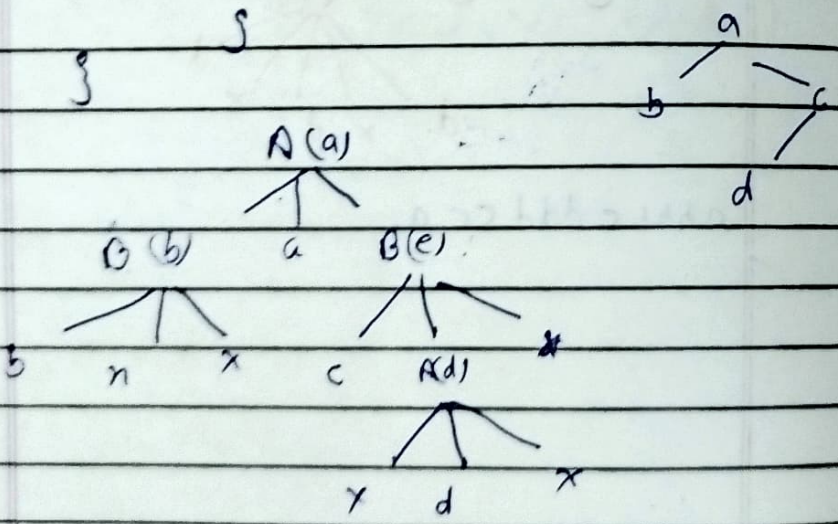
```
void What (Struct Node *t)
{
    if (t)
    {
        Printf ("%d", t -> data);
        What (t -> lc);
        Printf ("%d", t -> data);
        What (t -> Rc);
        Printf ("%d", t -> data);
```



a b b b c d d d c c a.

find what this function is doing

```
Void A (Struct Bnode *t)
{
    if(t)
    {
        B(t->LC);
        Printf("%d", t->data);
        B(t->RC);
    }
}
Void B(Struct Bnode *t)
{
    if(t)
    {
        Printf("%d", t->data);
        A(t->LC);
        A(t->RC);
    }
}
```

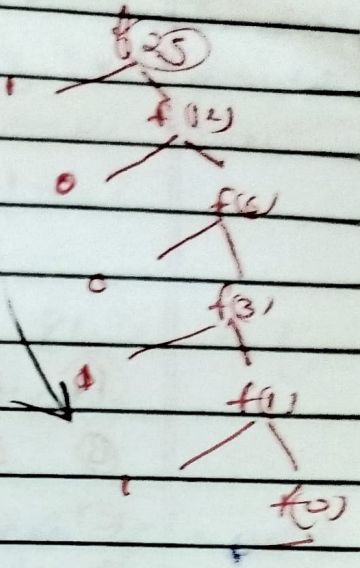


bacd

Date:- / /
Page:-

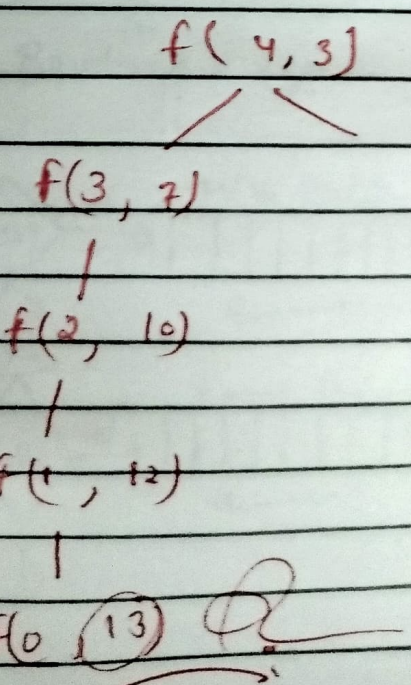
Q What does following fun Print n=25;

```
Void fun (int n)
{
    if (n==0)
        return;
    Printf ("%d", n%2);
    fun (n/2);
}
```



Q Recursive fun (x, y). What is the value of fun (4, 3).

```
int fun (int x, int y)
{
    if (x==0)
        return y;
    return fun (x-1, x+y);
}
```



Date: / /
Page:

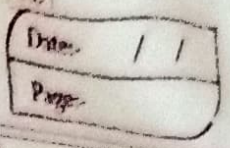
```
int fun (int x, int y)
{
    if (y == 0)
        return 0;
    return (x + fun(x, y-1));
}
```

(A) x+y

(B) x+x*y

(C) x*y

(D) x^y



Queue — FIFO

Queue is a linear list of elements in which deletions can take place only at one end called front.

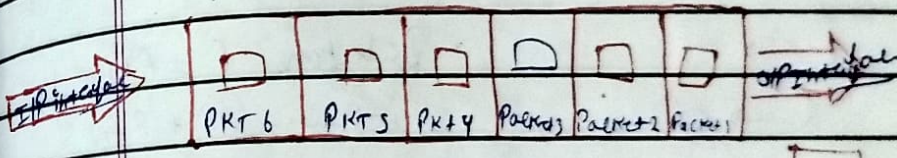
and

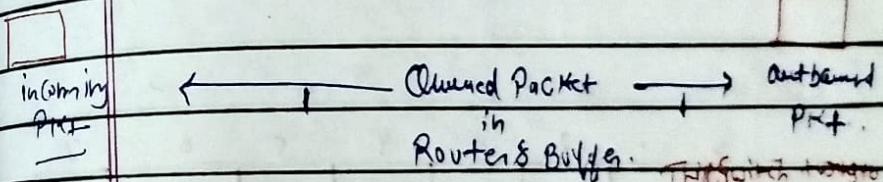
insertion can take place only at the end called rear.

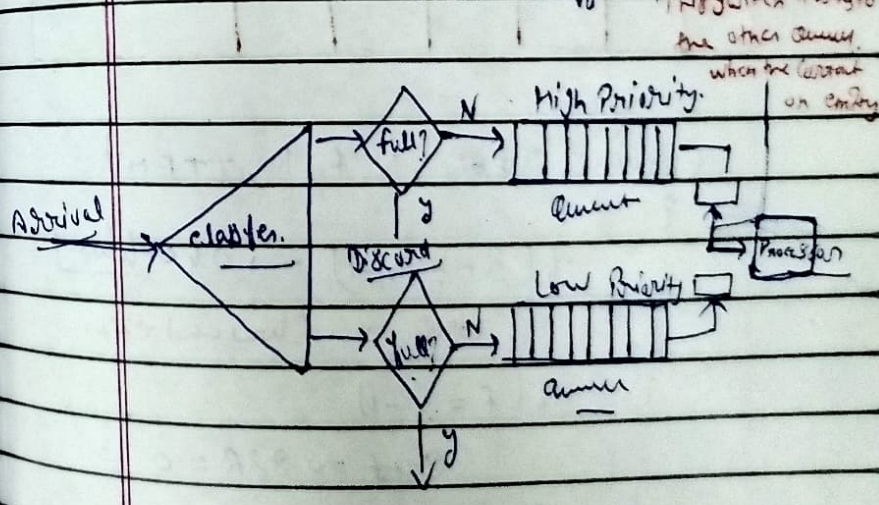
the terms front and Rear are used in describing a linear list.

4.6

A queue is a non Primitive linear data structure it is a homogenous collection of elements.





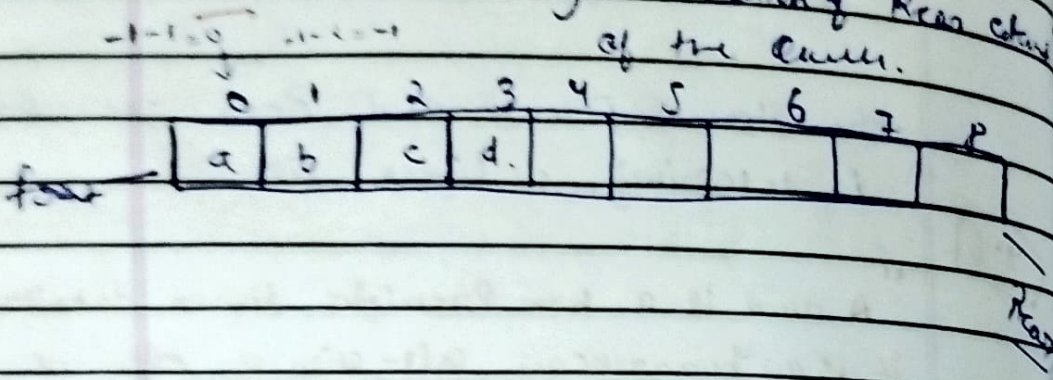


Date / /
Page

Mostly each of array queue will be maintained by a linear array QUEUE and two pointer variable.

front - Containing location of front element.

Rear — Containing location of Rear element of the queue.



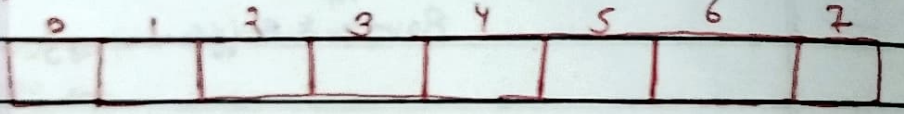
f → -1

R → -1

Deletion f = f+1

insertion R = R+1

Initialize —



Enqueue (QUEUE, N, f, R, ITEM)

?

if (R == N-1) → overflow

write overflow and exit.

if (f == -1)

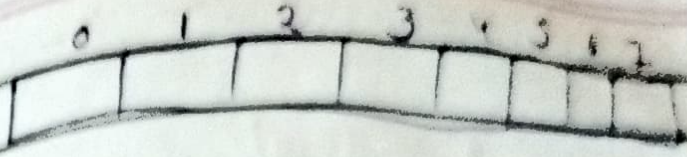
Set f = 0 & R = 0

else

R = R + 1

Queue[R] = ITEM.

5



Dequeue (Queue, N, f, R, ITEM)

{

if (f == -1)

Write underflow Stmt

ITEM = QUEUE[f]

if (f == R)

Set F = -1 & R = -1

else

f = f + 1

Return item

}

```
struct Queue
{
    int arr[MAX_SIZE];
    int front;
    int rear;
};
typedef struct Queue Queue;

void initialize(Queue *q)
{
    q->front = -1;
    q->rear = -1;
}

int isEmpty(Queue *q)
{
    return q->front == -1;
}

int isFull(Queue *q)
{
    return q->rear == MAX_SIZE - 1;
}
```

Date: / /
Page:

```
void enqueue (Queue *q, int item)
{
    if (isfull(q))
    {
        Printf ("Queue is full \n");
        return;
    }
    if (isEmpty (q))
    {
        q -> front = 0;
    }
    q -> arr [++(q -> rear)] = item;
}

int dequeue (circularQueue *q)
{
    if (isEmpty (q))
    {
        Printf ("Queue is empty! \n");
        exit (1);
    }
    int dequeuedItem = q -> arr[q -> front];
    if (q -> front == q -> rear)
    {
        q -> front = -1;
        q -> rear = -1;
    }
    else
    {
```

Date / /
Page

q->front = (q->front+1)%MAX_SIZE;
}
return dequeued;
}
#include<stdio.h>
#include<stdlib.h>
#define MAX_SIZE 100
int main()
{
Queue q;
initialize(&q);

return 0;
}

Queue with the help of linked list -

typedef struct Node
{
int data;
struct Node *next;
}Node;
typedef struct
{
Node *front;
Node *rear;
}Queue;

Insertion / deletion same end at Stack
" " diff end at " Queue.

Date: / /
Page:

```
Void enqueue (Queue *q, int item)
{
    Node * newNode = (Node*) malloc(sizeof(
    if (newNode == NULL)
    {
        Printf("Queue overflow \n");
        return;
    }
newNode → data = item;
newNode → next = NULL;
if (isEmpty(q))
{
    q → front = newNode;
}
else
{
    q → rear → next = newNode;
}
q → rear = newNode;
}
```

```
int dequeue(Queue* q)
{
    if (isEmpty(q))
    {
        Printf("Queue underflow\n");
        exit(1);
    }
    Node* temp = q->front;
    int dequeuedItem = temp->data;
    q->front = q->front->next;
    if (q->front == NULL)
    {
        q->rear = NULL;
    }
    free(temp);
    return dequeuedItem;
}

#include <stdio.h>
#include <stdlib.h>
int main()
{
    Queue q;
    initialize(&q);

    return 0;
```

Date: / /
Page:

Q Consider the following sequence of operation on an empty stack.
Push(54); Push(52); Pop(); Push(55);
Push(62); s = Pop();

Q Consider the following sequence of operation on an empty queue. enqueue(21); enqueue(24); dequeue(); enqueue(28); enqueue(32); q = dequeue(); The value of s + q is

62
55
~~52~~
54

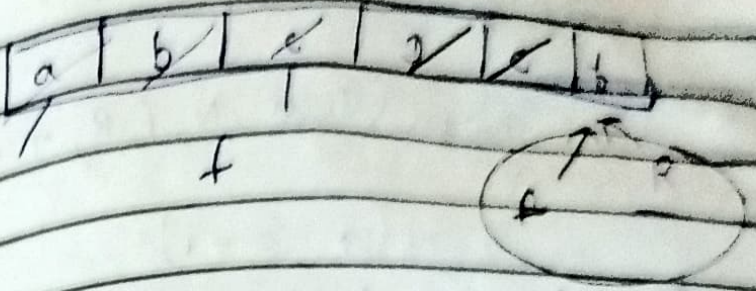
Pop

S = 62

21	24	28	32

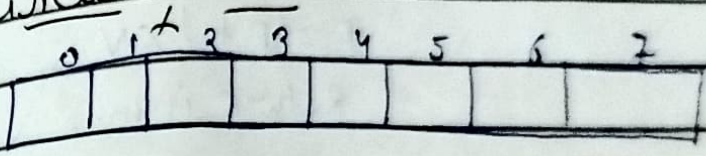
q = 24

S = 62
q = 24
86

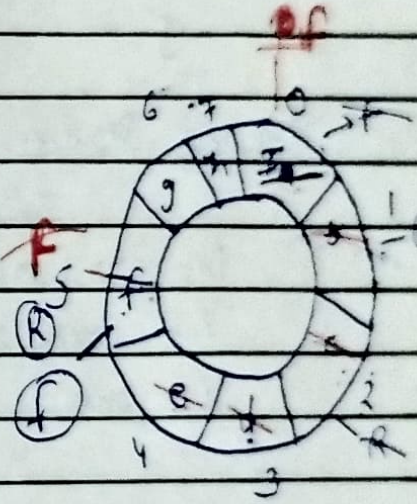


Problem
So we use
Circular
Queue

Circular Queue :—



En Q a, b, c
De Q 1 element
En Q d, e, f.
En Q g, h, i
De Q 4 element
En Q j, k, l, m, n.



Insertion :—

```
Enqueue (QUEUE, N, F, R, ITEM)
{
    if (f=0 && R==N-1) || (f==R+1)
        Write overflow and exit.
    if (f==-1)
        Set f=0 && R=0
    else
        R = (R+1) % N
    Queue[R] = Item.
}
```

Date / /
Page

Deletion :—

Dequeue (Queue, N, F, R, ITEM)
{
if (F == -1)
write Under flow and exit
ITEM = QUEUE [F]
if (F == R)
Set F = -1 && R = -1
else
F = (F+1) % N.
Return item.
}

typedef Struct
{
int arr [MAX_SIZE];
int front;
int rear;
}
Circular Queue;

Void initialize (circularQueue *q)
{
q → front = -1;
q → rear = -1;
}

```
int isEmpty (circularQueue *q)
{
    return q->front == -1;
}
int isFull (circularQueue *q)
{
    return (q->rear+1) % MAX_SIZE ==
                                q->front;
}

void enqueue (circularQueue *q, int item)
{
    if (isFull(q))
    {
        Printf ("Queue is full");
        return;
    }
    if (isEmpty(q))
    {
        q->front = 0;
        q->rear = 0;
    }
    else
    {
        q->rear = (q->rear+1) % MAX_SIZE;
    }
    q->arr[q->rear] = item;
}
```

```
int dequeue(circularQueue *q)
{
    if(isEmpty(q))
    {
        printf("Queue is empty\n");
        exit(1);
    }
int dequeuedItem = q->arr[q->front];
if(q->front == q->rear)
{
    q->front = -1;
    q->rear = -1;
}
else
{
    q->front = (q->front+1) %MAX;
}
return dequeuedItem;
}

int main()
{
    CircularQueue q;
    initialize(&q)

}   return 0;
```

Queue using linked list

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

typedef struct
{
    Node *front;
    Node *rear;
} circularQueue;

void initialize(circularQueue *q)
{
    q->front = NULL;
    q->rear = NULL;
}

int isEmpty(circularQueue *q)
{
    return q->front == NULL;
}
```

Enqueue an item to the queue —

```
Void enqueue (CircularQueue *q, int item)
{
    Node *newNode = (Node*) malloc (sizeof(Node));
    if (newNode == NULL)
    {
        Printf ("Queue overflow\n");
        return;
    }
    newNode -> data = item;
    newNode -> next = NULL;
    if (isEmpty (q))
    {
        q -> front = newNode;
        q -> rear = newNode;
        newNode -> next = newNode;
    }
    else
    {
        newNode -> next = q -> front;
        q -> rear -> next = newNode;
        q -> rear = newNode;
    }
}
```

Date:- / /
Page:-

Dequeue an item from the queue

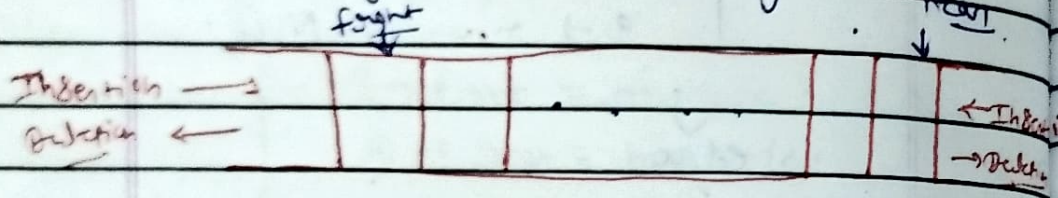
```
int dequeue (circularQueue * q)
{
    if (isEmpty (q))
    {
        Printf ("Queue Underflow \n");
        exit (1);
    }
    Node *temp = q -> front;
    int dequeuedItem = temp -> data;
    if ( q -> front == q -> rear)
    {
        q -> front = NULL;
        q -> rear = NULL;
    }
    else
    {
        q -> front = q -> front -> next;
        q -> rear -> next = q -> front;
    }
    free (temp);
    return dequeuedItem;
}
#include <stdio.h>
#include stdlib.h>
int main()
{
    CircularQueue q;
    initialize (&q);
```

Deque → insertion / deletion operations are performed at either end of the queue. We can insert an element from the rear end as the front end also deletion is possible from either end.

(*) Deque can be used both as a stack and as a queue.

Representation of Deque :—

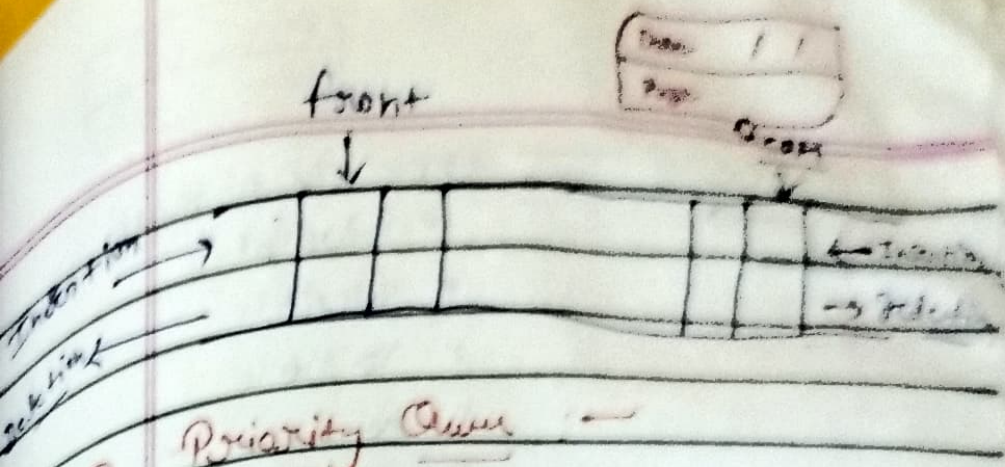
* using a doubly linked list
* using a circular array.



Types of deque :—

• Input - restrict deque :— element can be added at only one end but we can delete element from both end.

• Output - restrict deque — deletion take place at only one end but allows insertion at both ends.



Priority Queue

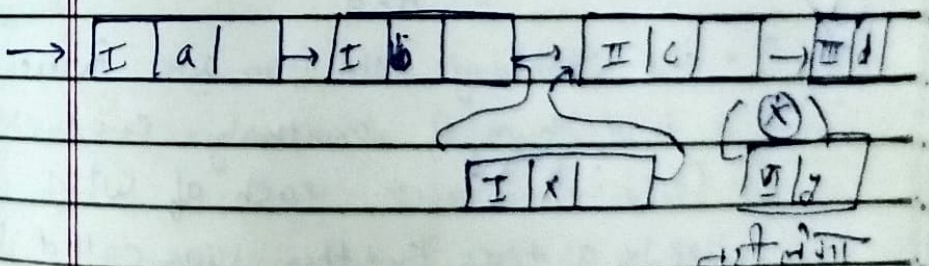
* is a collection of element with that each element has been assigned a Priority.

We have some rules for deleting/processing the element.

- An element of higher priority is processed before any element of lower priority.

- Two element with the same priority are processed according to the order in which they were added to the queue.

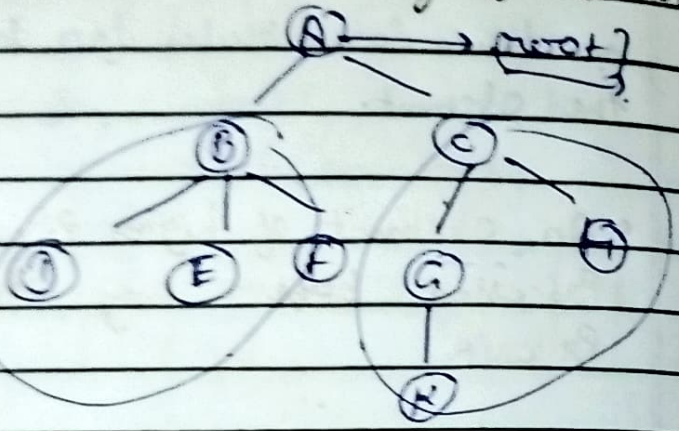
Class I, II III



Tree ⟶ tree is a most
* Powerful
* flexible
* versatile
* nonlinear
* advanced

data structure. It represents hierarchical relationship existing b/w several data item.

Used in wide range of application.



A tree is a finite set of one or more data item (nodes) such that:

- There is a special data item called root of the tree.

And.

- its remaining data item are partitioned into num of mutually exclusive (disjoint) subset, each of which is itself a tree and they are called subtree.

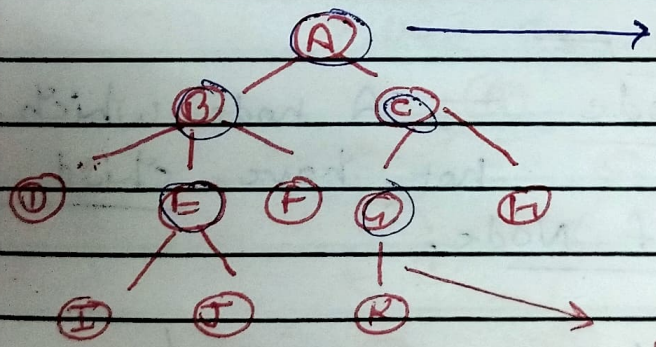
- Every node (exclude root) is conn" by a

Date:- / /
Page:-

directed edge from exctly one other node. A direction : Parent to Child.

Terminology :—

* **Root** :— * first/TOP most Node is called root Node.
* We always have exactly one root node in every tree.
* We can say that root node is the origin of tree data structure.



Here 'A' is the "root" node.

Edge

* **Edge** :— Connection link b/w any two nodes is called as Edge.

Parent का Child का Connect करने वाला.

* **Parent** :— The node which is Predecessor of any node is called as Parent Node.

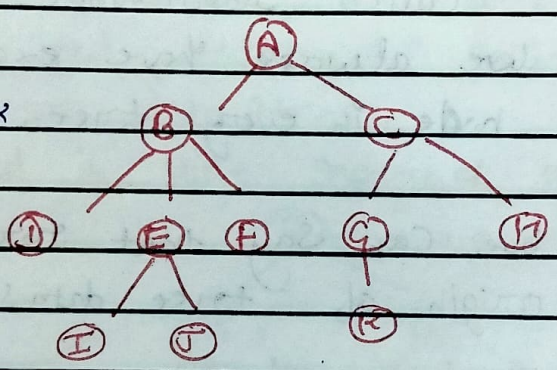
Any node which has children/child called Parent.

Ex :— A, B, C, E, G.

Date:- / /
Page:-

* Child Node :— The node which is descendant of any node is called as child Node.

* A Node which has the link from its Parent node is called as child node.



B, C, G, H, K

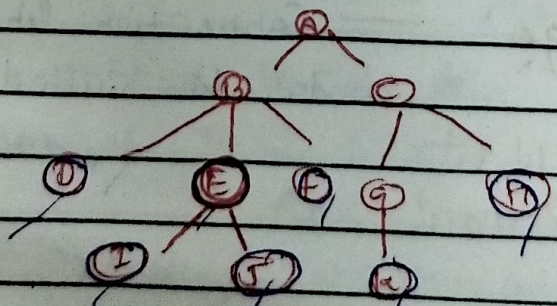
B's → D E F

E's - I, J -

External :—

* Leaf Node :— A node which does not have child is called as LEAf Node.

* Also called external node. / Terminal node.



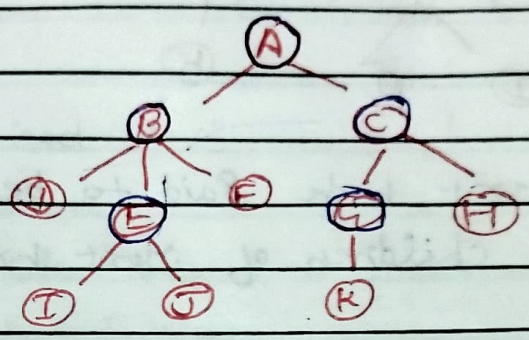
Here is D, I, J, K, F, & H are leaf node.

* A node without Successor is called leaf node.

leaf node का उल्टा Internal node

Date:- / /
Page:-

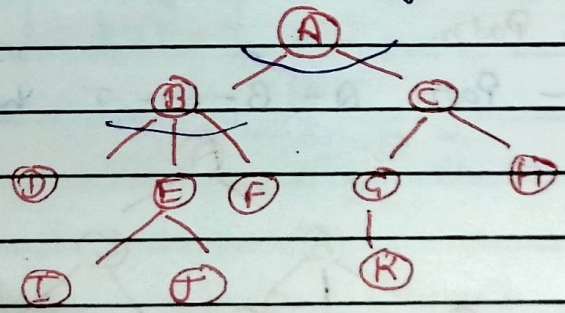
* Internal Node :— * A node which has at least one child is called as INTERNAL Node.



A, B, C, G, E

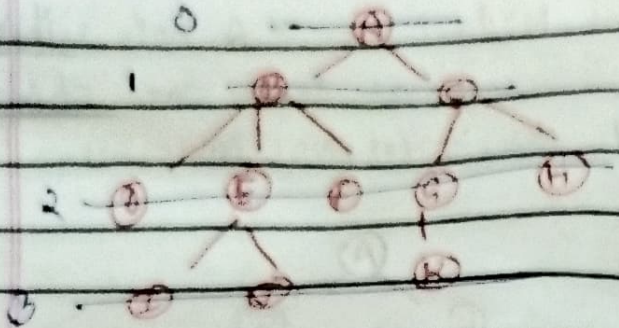
* node other then leaf node. Called Internal

* Degree :— Total number of children is called Degree.



Degree of A is 2.
Degree of B is 3
" " F is 0.

* Level / Depth / Height :— In tree each step from top to bottom is called as a level and the level count start with 0. and increment by one at each level.

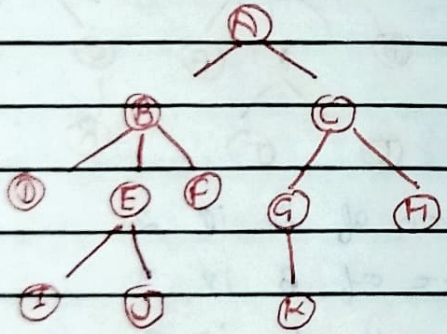


* root node said to be at level 0 and the children of root node are at level 1

* Path — The Sequence of Nodes / Edges from one node to another node is called as Path.

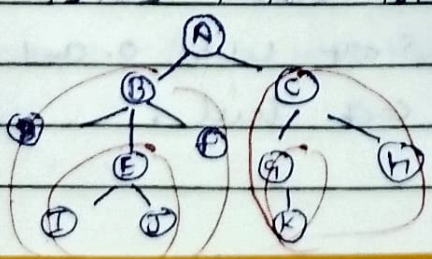
Length of a Path is total num of edges in that Path

— Path A - B - E - J has length 4.



A Path is the Sequence of nodes and edges edges b/w two nodes.

* Subtree — tree inside tree is called

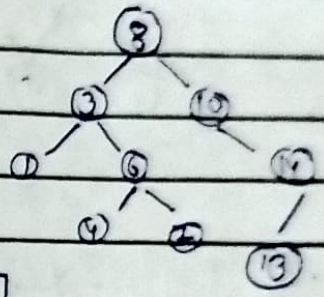


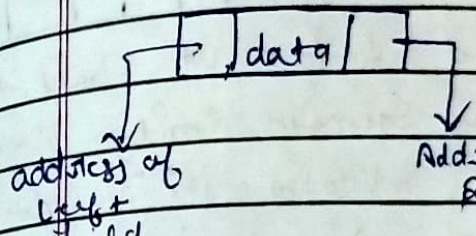
Date: / /
Page:

Binary Tree

A Tree T in which any node can have maximum two children (left / Right).

```
Struct node {
int data;
Struct node * left;
Struct node * right;
}
```





Q Let T be a binary search tree with 15 nodes. The minimum and maximum Possible height of T are -

(A) 4 and 15 respectively.
(B) 3 and 14 respectively
(C) 4 and 14 respectively.
(D) 3 and 15 respectively.

Ans: 2

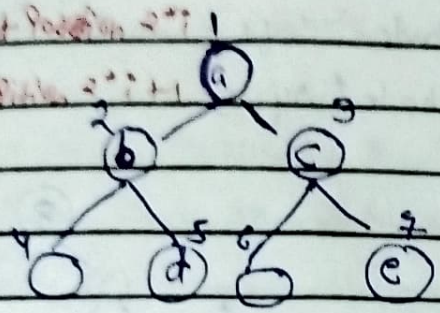
Binary tree representation using array

Binary tree can be represented using array

Root is at index 1

for any given node at Position i

- Left child is at Position $2*i$
- Right child is at Position $2*i+1$



1	2	3	4	5	6	7
A	B	C		D		E

If node does not have left / Right child, the position in the array remains empty. it filled with a special value indicating its vacant (like null or -1).

Linked Representation of binary trees :—

* Consider a binary tree T which used three parallel array, INFO, LEFT, RIGHT and a Pointer Variable ROOT

* first of all, each node N of T wi to a location K such that:
  - (a) INFO [K] contains the data at the
  - (b) LEFT [K] " " location of the chi node N.

Date:- / /
Page:-

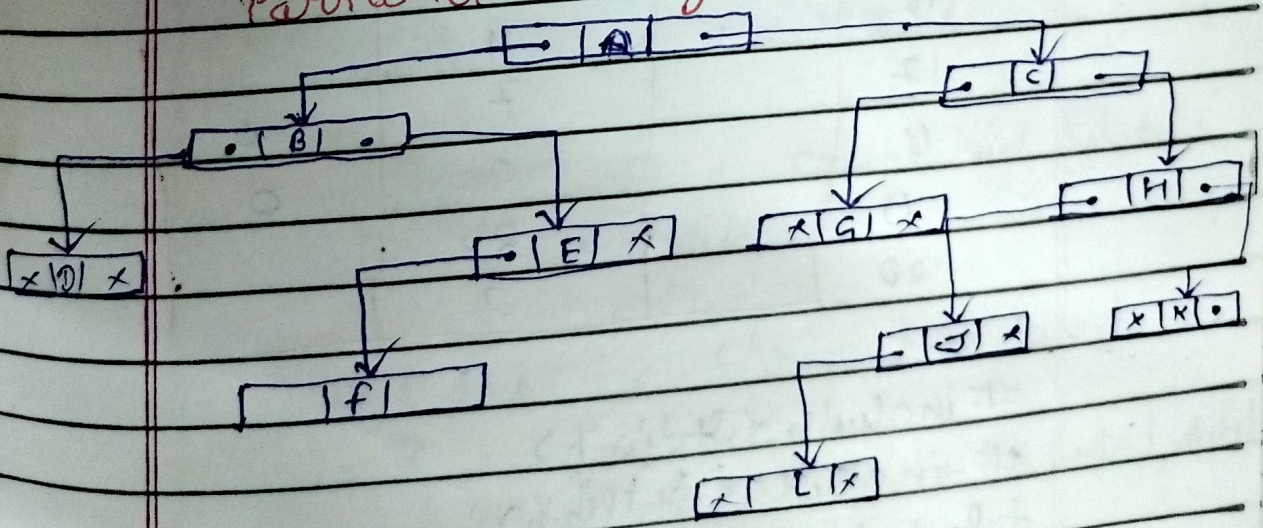
2. RIGHT[K] contains the location of the right child of node N.

3. ROOT will contains the location of the root R and T.

4. if any subtree is empty, then the corresponding pointer will contain null value.

5. if the tree T itself is empty, then ROOT will contain the null value.

6. INFO may actually be a linear array of record or a collection of parrallel arrays.



Date:- / /
Page:-

	INFO	Left	Right
1	K	0	0
2	C	3	6
3	G	0	0
4		14	
5	A	10	2
6	H	17	1
7	L	0	0
8		9	
9		4	
10	B	18	13
11		19	
12	F	0	0
13	E	12	0
14		15	
15		16	
16		11	
17		7	0
18		0	0
19		20	
20		0	

Root [5]

Avail [8]

```
#include<stdio.h>
#include<stdlib.h>
typedef struct Node
{
    int data;
    struct Node *left;
    struct Node *right;
} Node;
```

Date:- / /
Page:-

```
Node * createNode (int data)
{
    Node * newNode = (Node *) malloc (sizeof (Node));
    if (!newNode)
    {
        Printf ("Memory error \n");
        exit (1);
    }
    newNode -> data = data;
    newNode -> left = NULL;
    newNode -> right = NULL;
    return newNode;
}
void insert (Node ** root, int data)
{
    if (*root == NULL)
    {
        *root = createNode (data);
        return;
    }
    if (data < (*root) -> data)
    {
        insert (&((*root) -> left), data);
    }
    else
    {
        insert (&((*root) -> right), data);
    }
}
```

Date:- / /
Page:-

Traversal of binary tree :-

The process of visiting (checking / updating) each node in a tree data structure, exactly once is called tree traversal.

They may be traversed in depth-first or breadth-first order.

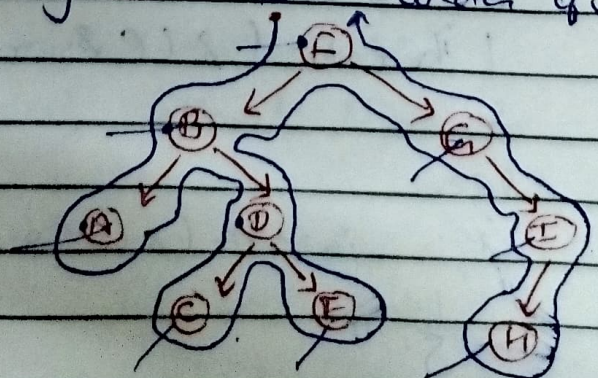
Way of traversed in depth-first.

* In-order — L Root R
* Pre order — Root L R
* Post order — L R Root.

* Pre-order (Root L R) :-

Pre-order : f, B, A, D, C, E, G, I, H.

* Check if the current node is empty / Null.
* Display the data part of the root
* Traverse the left subtree by recursively calling the Pre-order function.
* Traverse the right subtree by recursively calling the Pre-order function.

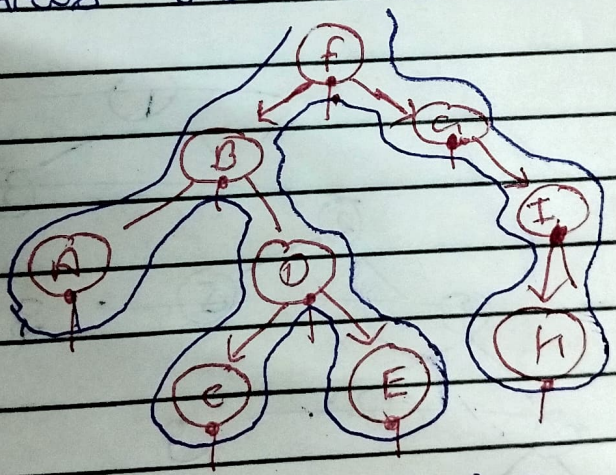


Date:- / /
Page:-

× in-order (L Root R)

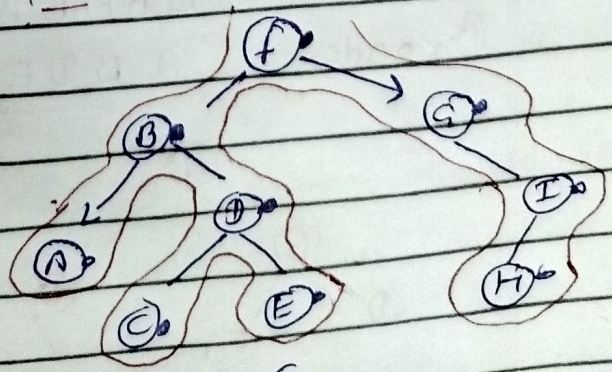
in-inorder A, B, C, D, E, F, G, H, I

- check if the current node is empty or null.
- Traverse the left subtree by recursively calling the in-order function.
- Display the data part of the root/Current node
- Traverse the right subtree by recursively calling the in order function.
- in binary search tree, in-order traversal retrieves data in sorted order.



A, B C, D, E F G, H, I

× Post order :-



A C, E D B, H I G F

Inorder → Sam L to R.

Post O → R to L.

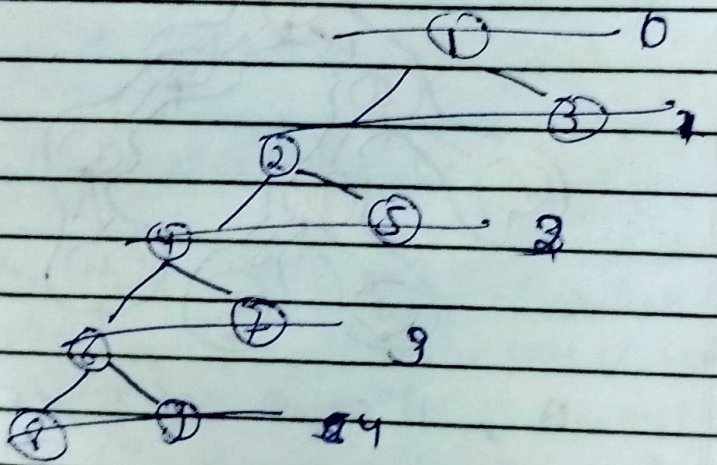
Pre O → L to R

Date: / /

Page:

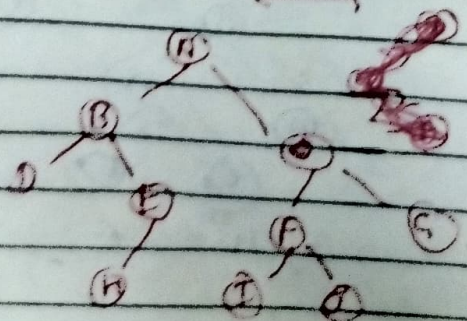
The Post order traversal of a binary tree is 8, 9, 6, 7, 4, 5, 2, 3, 1. The inorder traversal of the same tree is 8, 6, 9, 4, 7, 2, 5, 1, 3. The height of a tree is the length of the longest Path from the root to any leaf. The height of the binary tree above is ___

Pre T Post



Q) inorder : D B H E A I F J C G

Preorder : A B D E H C F I J G



Parent node left side first.
Right side next.

Date:- / /
Page:-

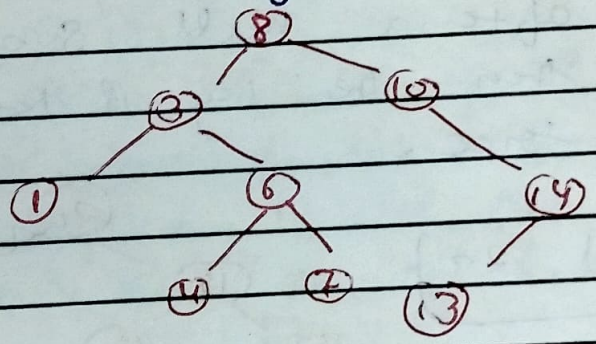
Binary Search tree / BST / ordered tree / Sorted tree.

Left Subtree of a node contains a key less than node's key.

or

Right Subtree of node contains a key greater than nodes key.

Left / Right Sub tree must each also be a binary search tree



In Binary Search Tree inorder traversal gives in sequence.

1, 3, 4, 6, 7, 8, 10, 13, 14

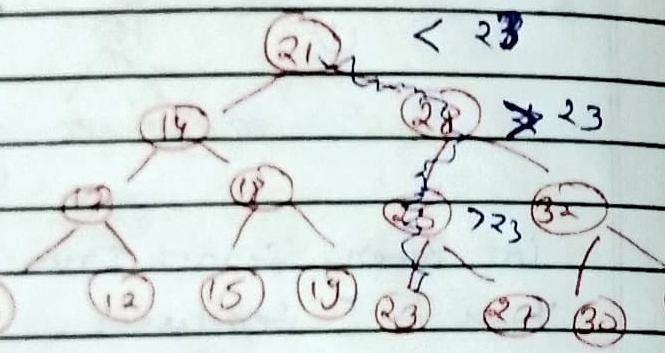
Searching → We start by examining the root node.

* if tree is null the key we are searching for does not exist in the tree, otherwise
* if the key equals that of the root, the search is successful and we return the node.

- if the key is less than that of the root, we search the left subtree.
- if the key is greater than that of the root we search the right subtree.
- This process is repeated until the key is found or the remaining subtree is null.

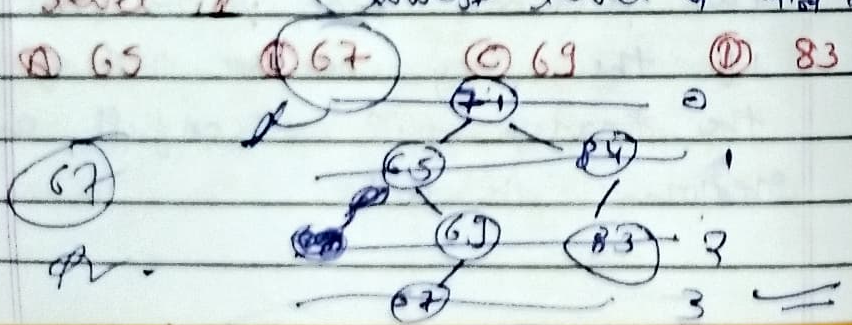
if the searched key is not found after a null subtree is reached then the key is not present in the tree.

find 23



Q While inserting the elements 71, 65, 84, 69, 67, 83 in an empty binary search tree (BST) in the sequence shown, the element in the lowest level is (lowest level में क्या आएगा)

(A) 65 (B) 67 (C) 69 (D) 83



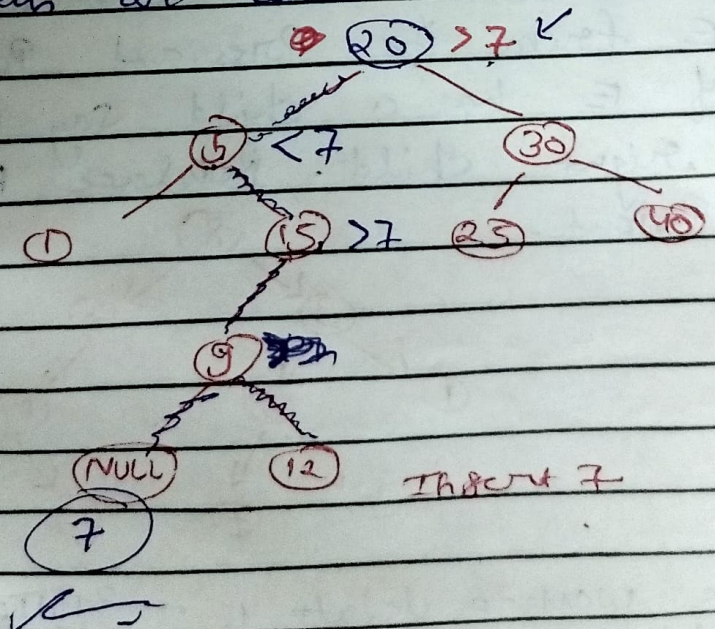
Key देखना है

Date:- / /
Page:-

Insertion — if the key is not equal the root we search the left or right subtree as before.

Eventually, we will reach an external node and add the new key value pair (here encoded as a record 'new Node') as its right or left child depending on the node's key.

In other words we examine the root and recursively insert the new node to the left subtree if its key is less than that of the root or the right subtree if its key is greater than or equal to the root.

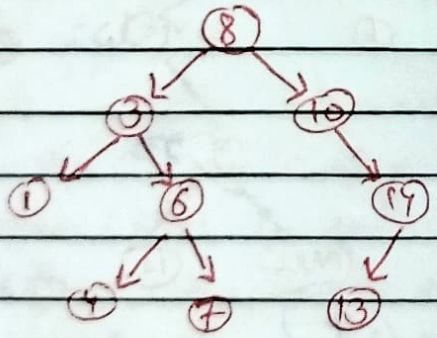


Deletion

- Deleting a node with no children: Simply remove the node from the tree.
- Deleting a node with one child. Remove the node and replace it with its child.
- Deleting a node with two children: Call the node to be deleted D. Do not delete D.

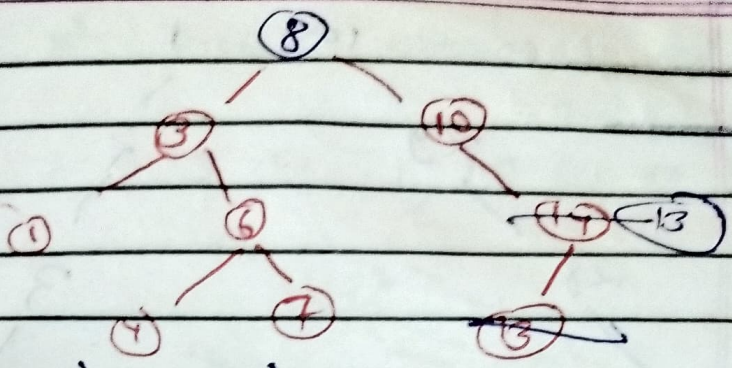
- Instead, choose either its in-order Predecessor node or its in-order Successor node as replacement node. E. Copy the user values of E to D.

- If E does not have a child simply remove E from its previous Parent G. If E has a child say F, it is a right child. Replace E with F at E's Parent.



imagin we want to delete 4 — अगर Node के left child नहीं होता तो उसे delete करना easy हो जाता है।

Date:- / /
Page:-



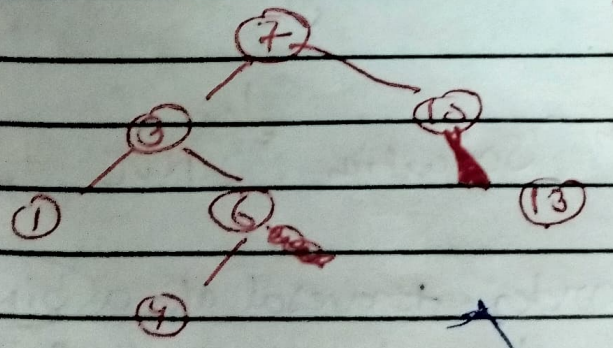
आगर हमें 14 को delete करना है तो 14 के नीचे 13 है तो 13, 14 से replace हो जायेगा।

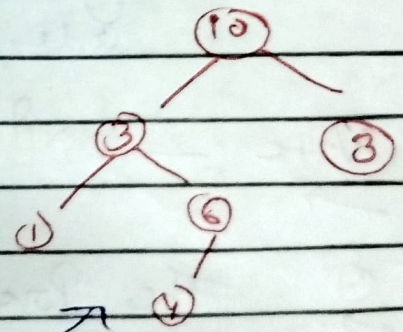
if we want to delete 8 — inorder predecessor

Shorted form - 1 3 6 7 8 10 13

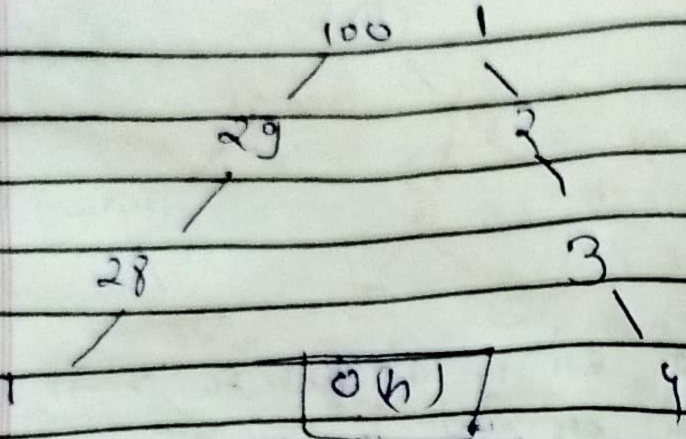
inorder successor

inorder Predecessor → किसी को भी replace कर सकते हो।
inorder Successor





Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n).



ये काम linked list stack queue से भी हो सकता है that is big disadvantage of BST.

The major advantage of binary search trees over other data structure is that the related sorting algorithms and search algorithm such as in order traversal can be very effective they are also easy to code.

Note :— इसी का solution है AVL tree

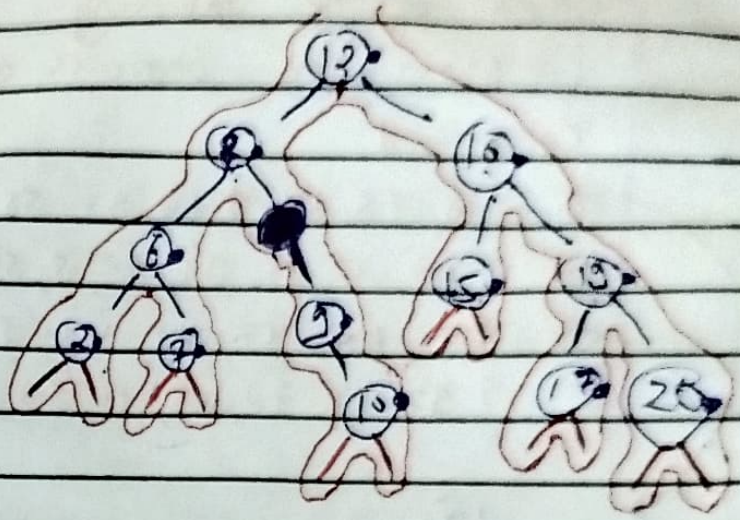
Q The Pre-order traversal of a binary search tree is given by 12, 8, 6, 2, 7, 9, 10, 16, 15, 19, 17, 20. Then the Post-order traversal of this tree is.

(a) 2, 6, 7, 8, 9, 10, 12, 15, 16, 17, 19, 20
(b) 2, 7, 6, 10, 9, 8, 15, 17, 20, 19, 16, 12
(c) 1, 3, 5, 7, 8, 15, 19, 25
(d) 4, 6, 7, 19, 18, 20, 25.

Date / /
Page

BST का inorder traversal sorted होता है।

2, 6, 7, 8, 9, 10, 12, 15, 16, 17, 18, 20



Post → 12, 8, 6, 2, 7, 9, 10, 16, 15, 19, 17, 20

Inorder → 2, 6, 7, 8, 10, 12, 15, 16, 17, 20

Post → 2, 7, 6, 10, 9, 8, 15, 17, 20, 19, 16, 12

Q Which of the following is/are correct inorder traversal sequence of binary search tree?
(1) 1, 3, 5, 7, 8, 15, 19, 25.
(2) 5, 8, 9, 12, 10, 15, 25.
2, 7, 10, 8, 14, 16, 20
4, 6, 7, 9, 18, 20, 25.
(a) 1 and 4 only (b) 2/3 only (c) 2 and 4 only
(d) 2 only

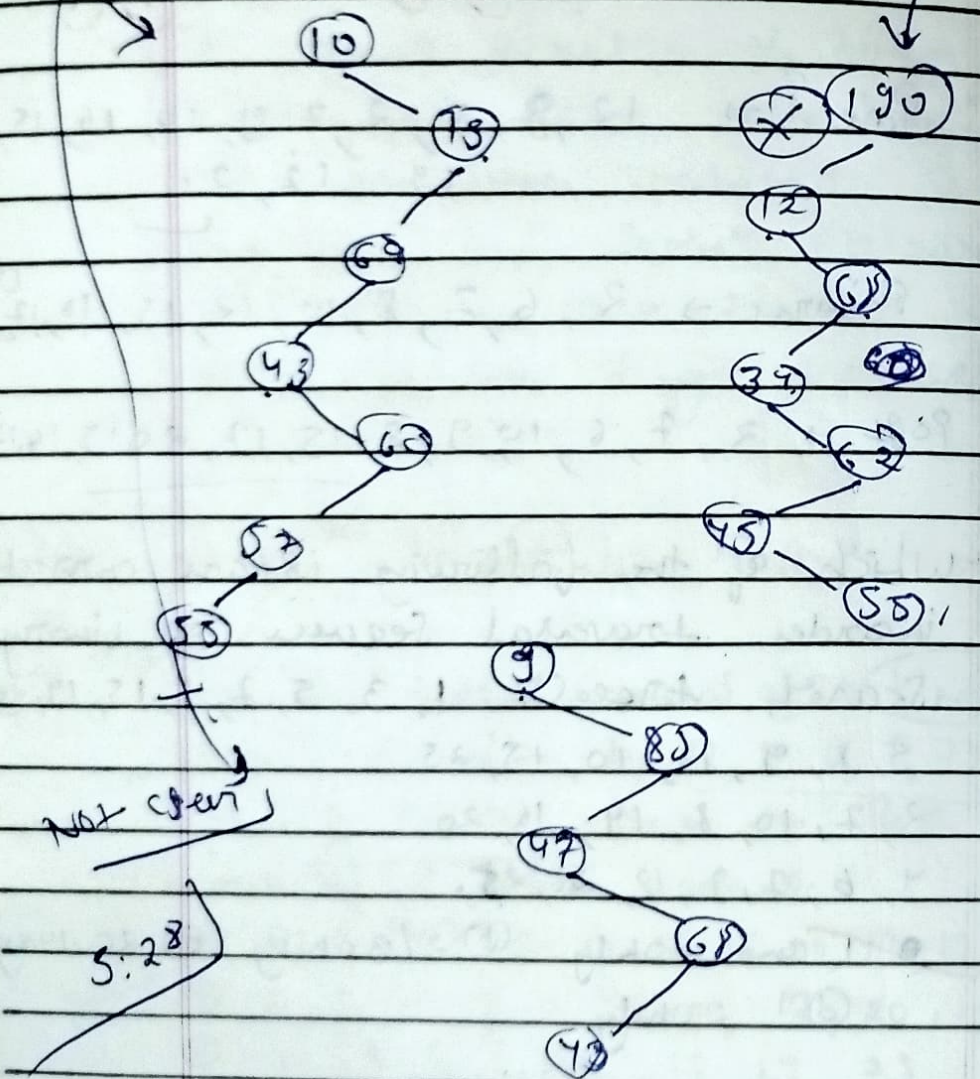
Q Suppose that we have numbers between 1 and 100 in a binary search tree and want to search for the number 55. Which of the following sequences could be the sequence of nodes examined

① {10, 75, 64, 43, 60, 57, 55}

② {90, 12, 68, 34, 62, 45, 55}

③ {89, 89, 47, 68, 43, 57, 55}

④ {97, 17, 72, 56, 16, 53, 55}.



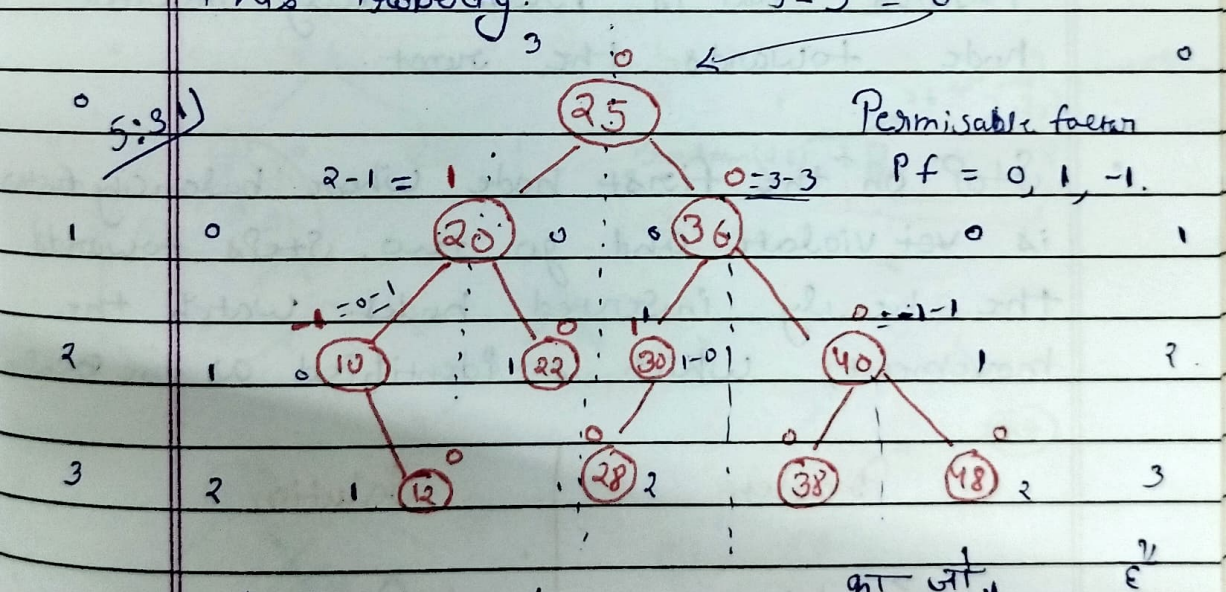
```
Node * SearchBST (Node* root, int data)
{
    if (root == NULL || root->data == data)
    {
        return root;
    }
    if (data < root->data)
    {
        return SearchBST (root->left, data);
    }
    return SearchBST (root->right, data);
}

Void insertBST (Node** root, int data)
{
    if (*root == NULL)
    {
        *root = createNode (data);
        return;
    }
    if (data < (*root)->data)
    {
        insertBST (&((*root)->left), data);
    }
    else if (data > (*root)->data)
    {
        insertBST ((*root)->right, data);
    }
    else {
        Print ("Element already exist in BST");
    }
}
```

```
Node* deleteBST(Node* root, int data)
{
    if (root)
        return root;
    if (data < root->data)
    {
        root->left = deleteBST(root->left, data);
    }
    else if (data > root->data)
    {
        root->right = deleteBST(root->right, data);
    }
    else
    {
        if (root->left)
        {
            Node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right)
        {
            Node* temp = root->left;
            free(root);
            return temp;
        }
        Node* temp = findValueNode(root->right);
        root->data = temp->data;
        root->right = deleteBST(root->right, temp->data);
    }
    return root;
}
```

AVL Tree — An AVL tree (named after inventors Adelson-Velsky and Landis) is a self-balancing binary search tree.

* "The heights of the two child subtrees of any node differ by at most one.

* if at any time they differ by more the one, rebalancing is done to restore this Property."



Left Sub tree / Right Sub tree

diff calculate

* Balance factor :—

In binary tree the balance factor of a node N is defined to be the height diff balance factor (N) = Height (Left Subtree (N)) - Height (Right Subtree (N)) of its two child subtrees.

A binary tree is defined to be an AVL tree if the invariant - Balance factor (N) ∈ {-1, +1} holds for every node N in the tree.

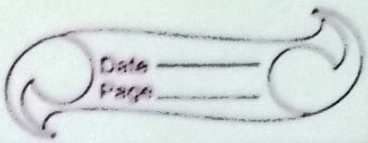
Insertion in an AVL tree

- Insert a node similarly as we do in BST.
- After the insertion start checking the balancing factor of each node in a bottom up fashion that is from newly inserted node towards the root.
- Stop on the first node whose balancing factor is not violate and go two steps towards the newly inserted nodes. Watch the movement, which is identified as the Problem.

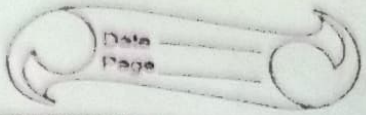
Problem	Solution
✓ LL	R ←
→ RR	L
LR	LR
RL	RL

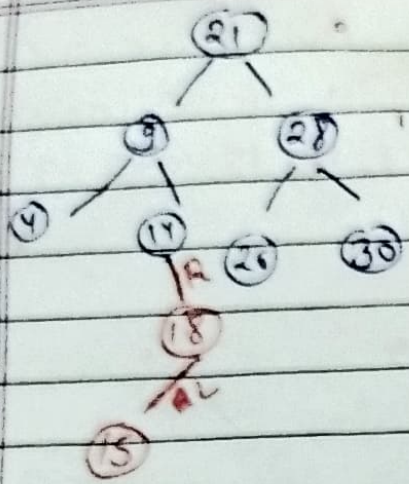
AVL tree ही BST ही एक BST.

S:36



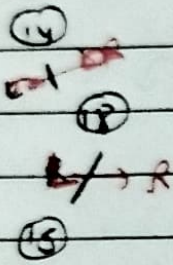
Q Consider an empty AVL tree and insert the following nodes in sequence.
21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7 ;





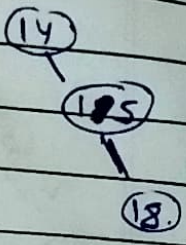
LHS = 3
RHS = 2 = 1

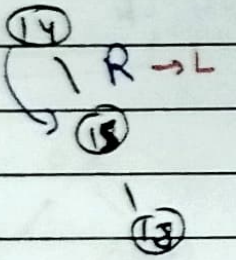
LHS = 4
RHS = 2 = 2 Problem
RL = RL



नीचे से R फेर दो

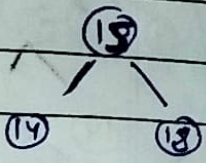
=

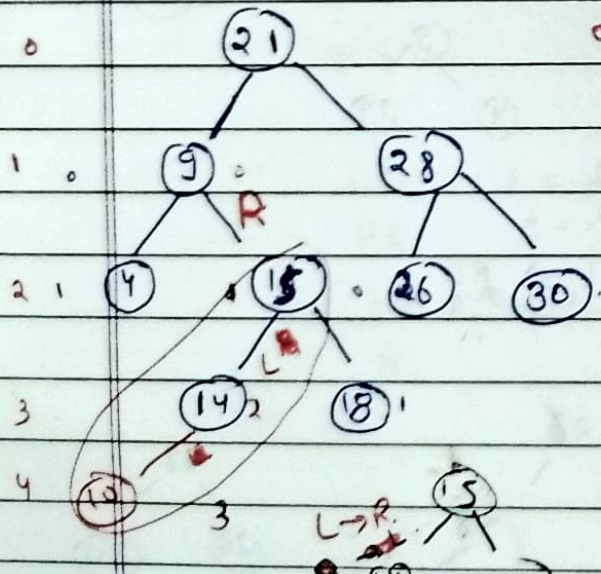




RL फेर दो.

⇒

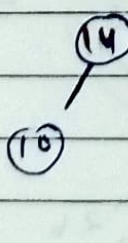


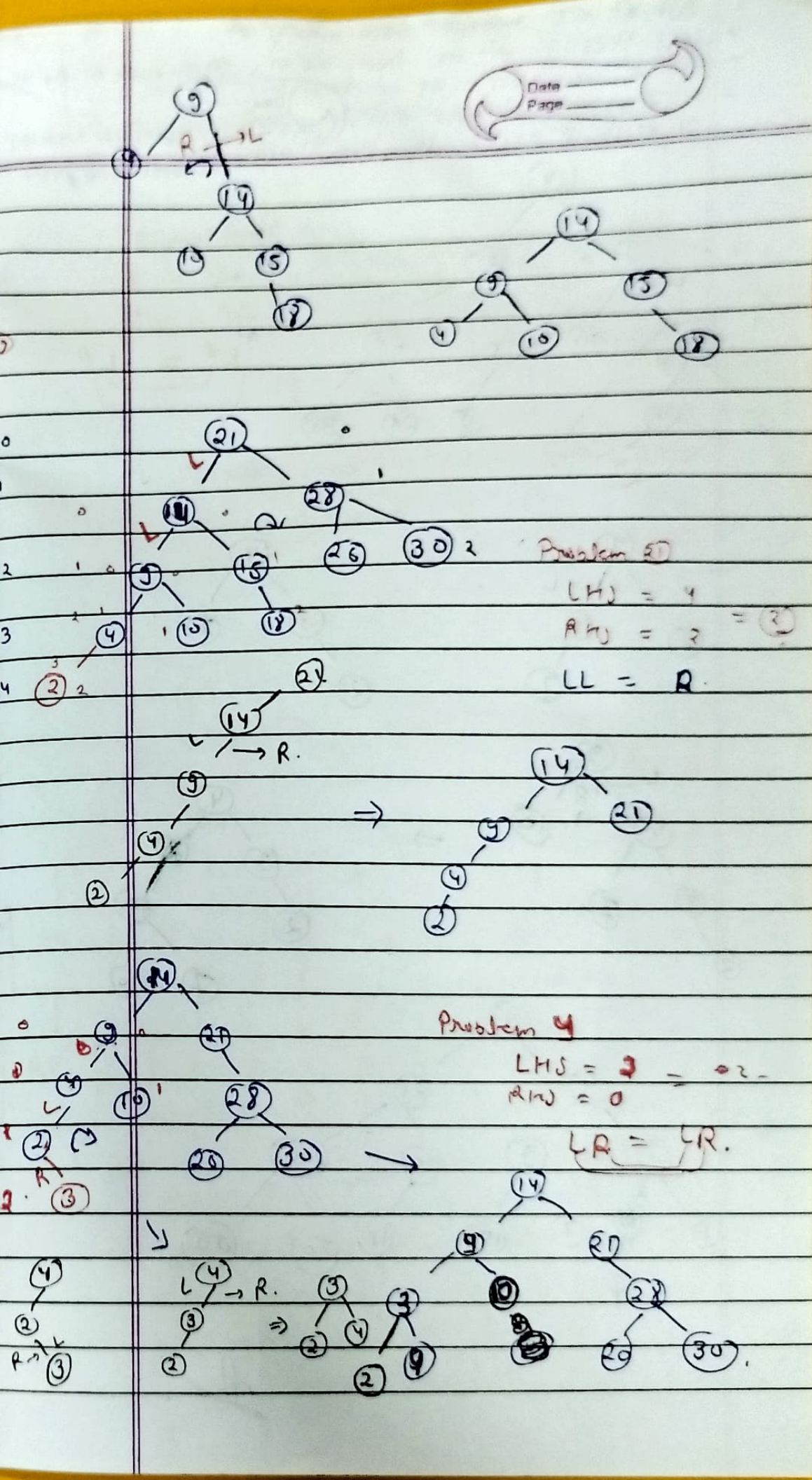


Problem (9, 21) —

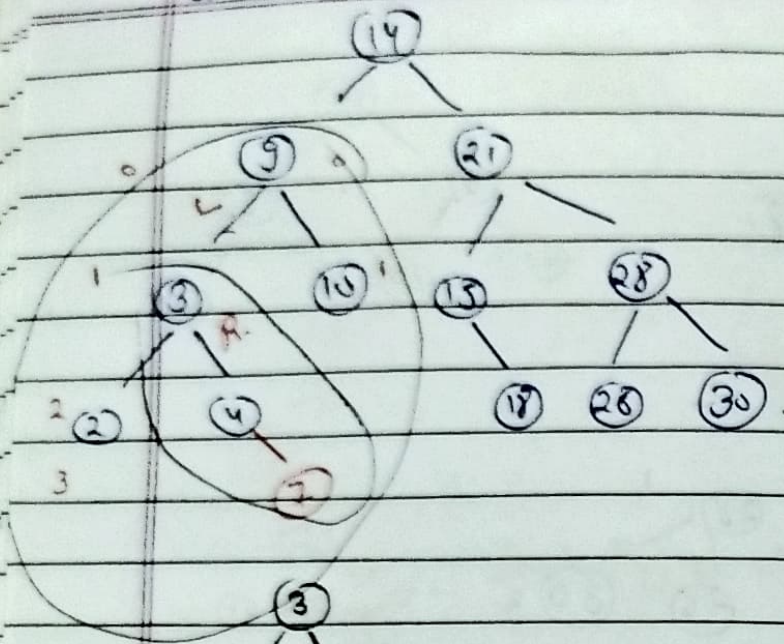
LHS = 1
RHS = 3 = -2

LR = LR




Problem 5
LHS = 4
RHS = 2 = 2
LL = R
Problem 4
LHS = 2 = 2
RHS = 0
LR = LR.

* इसे ...
* जहां Position हो वहां insert कर दो
* जहां से insert करा है वहां से check करो कि किसी Node का Bf ...
* अगर disturb हुआ है तो पहले वाले पे आओ और जिस Node का Bf ख़राब है वहां से ऊपर चलो ... पहले दो Node देखो पर

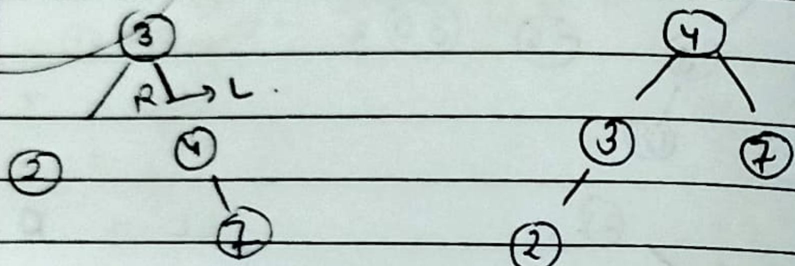


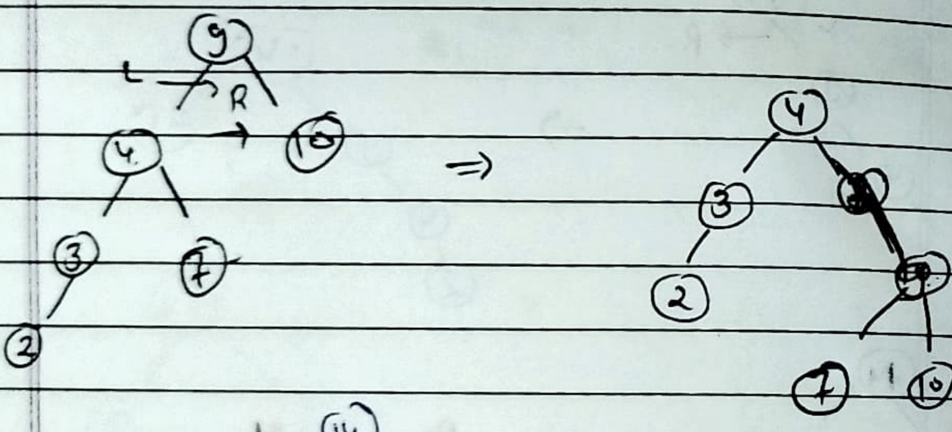
Problem 9

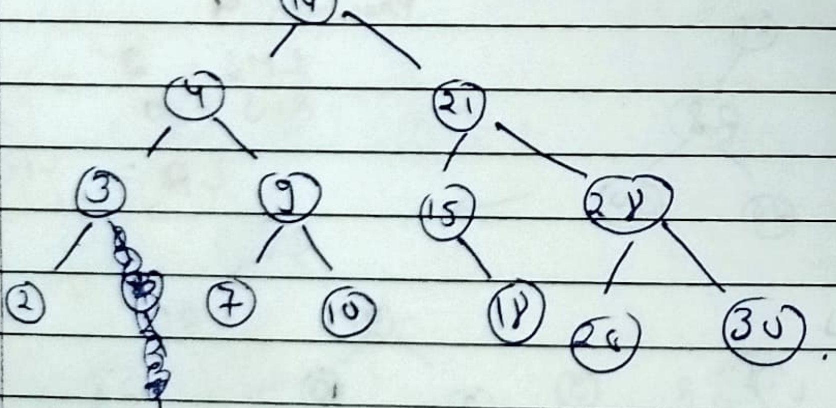
L = 3

R = 1

LR = LR





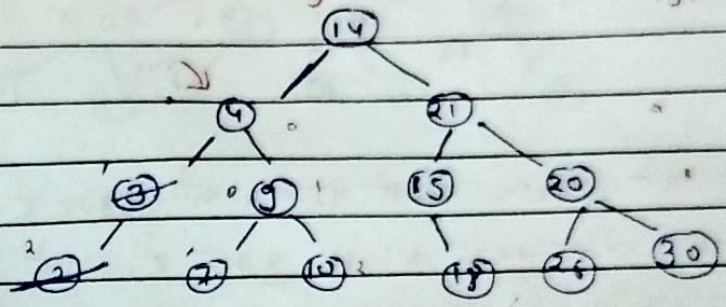


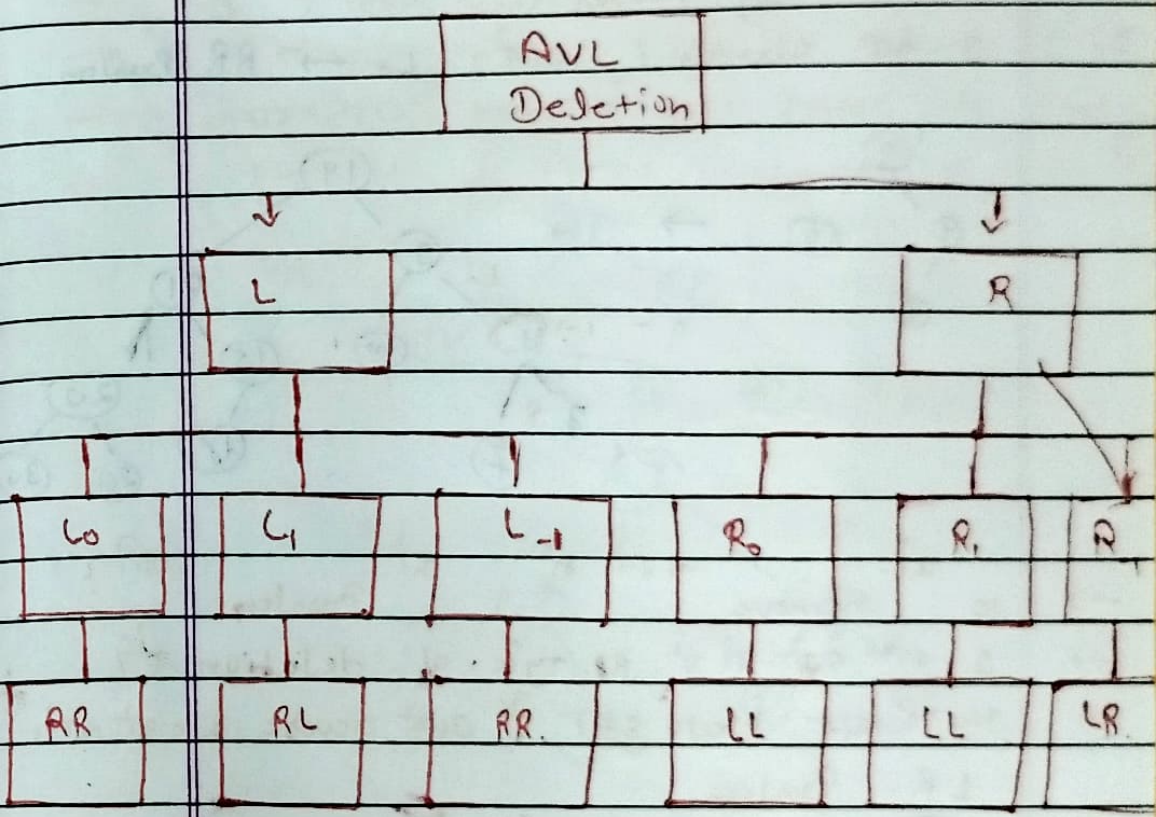
* No LL, RR, LR, RL ... according Rotation ...

* tree worst case ... log(h) time ...

Deletion in an AVL tree :—

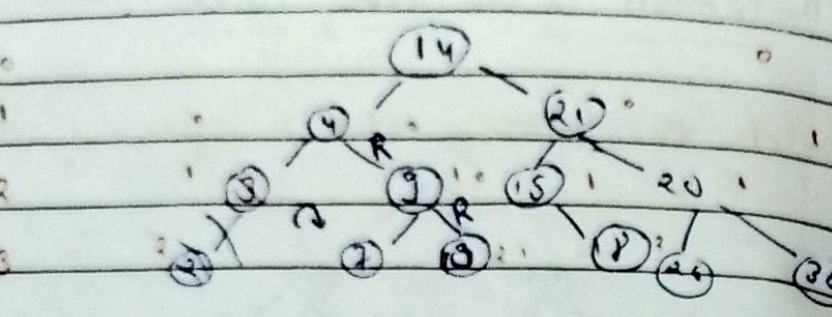
Q → Delete the following nodes in sequence 7, 3, 10, 8, 4, 2, 14, 1





[One more time learn] →

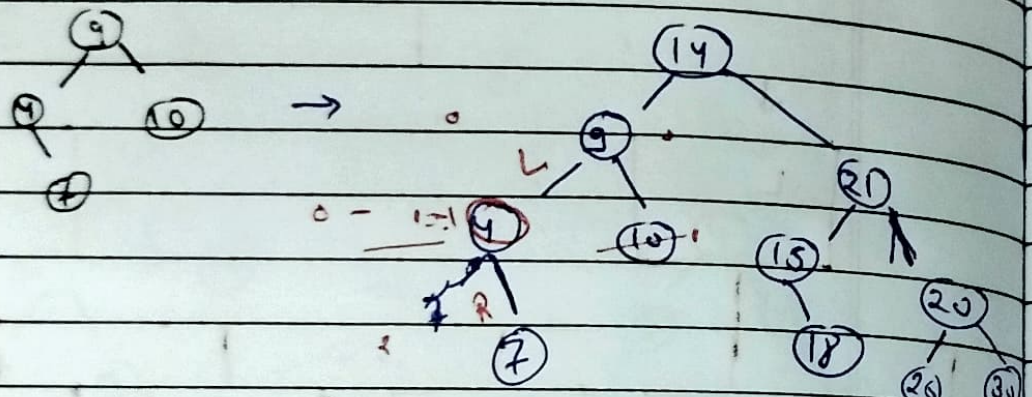
2 3 10 18 4 9 14 7 15



→ 3 को delete करने पर 4 को issue है।

→ अब जिस node delete हुआ है उसके Reverse side का Balancing factor check करना है

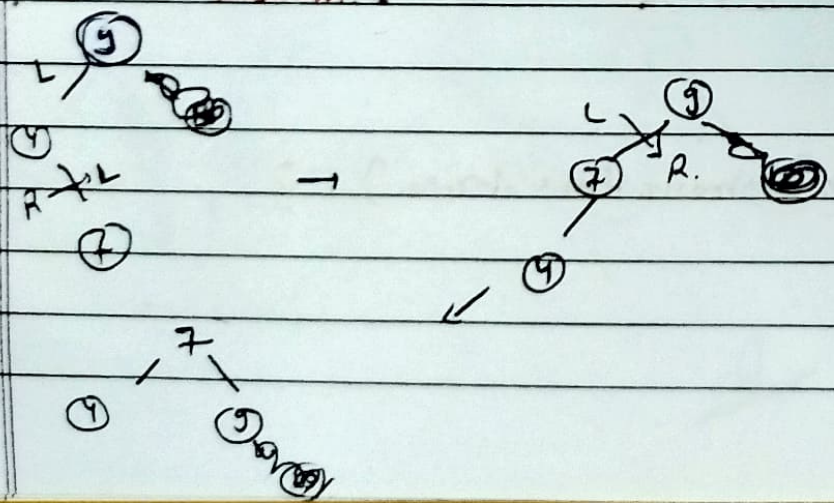
→ 9 का Balancing f 0 है। Lo → RR Problem.

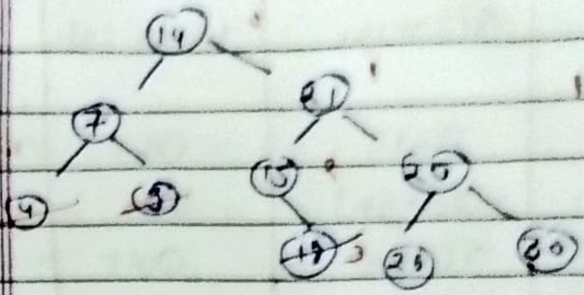


→ 10 को remove करने में 9 का problem होगी!

→ 9 को देखो तो पर R. type of deletion है।

→ अब जिस delete हुआ है उसके reverse side को मोड़ो LR Problem.





1 − 2 = −1

18 delete करने में कोई Problem नहीं है।

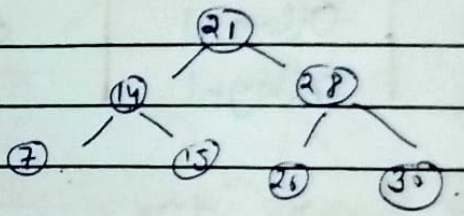
4 " " " " " " "

→ 9 को delete करने में 14 में Problem होगी।

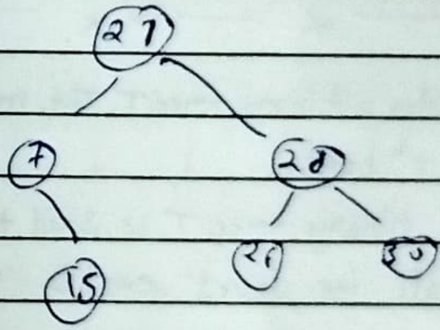
→ 14 से देखो तो ये L type का deletion है

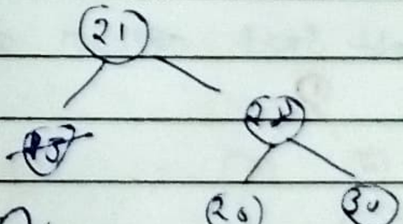
→ जिधर delete हुआ है उसके Reverse side के child

1 − 2 = −1 तो ये L-1 Problem है तो ये RR है

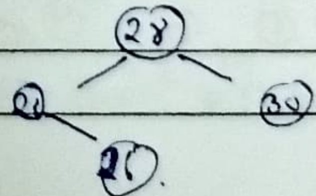


R_{-1}





RR Problem. L R



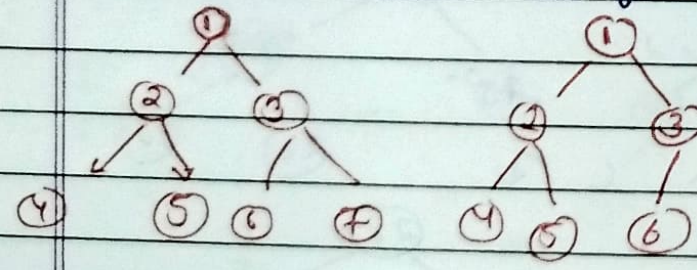
Algorithm	Average	Worst Case	
Space	O(n)	O(n)	Binary Search tree
Search	O(log n)	O(n)	
Insert	O(log n)	O(n)	
Delete	O(log n)	O(n)	

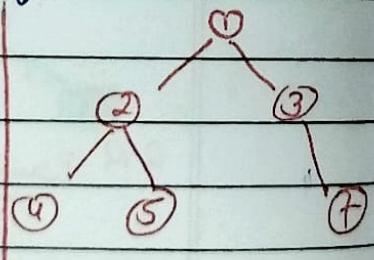
Algorithm	Average	Worst Case	
Space	O(n)	O(n)	AVL tree
Search	O(log n)	O(log n)	
Insert	O(log n)	O(log n)	
Delete	O(log n)	O(log n)	

* **Complete Binary tree** :— L to R, T to D.

* Consider a binary tree T, The maximum num of nodes at height 'h' is 2^h nodes.

* The binary tree T is said to be complete binary tree, if all its level except possibly the last, have the maximum num of nodes and if all the nodes at the last level appear as far left as possible



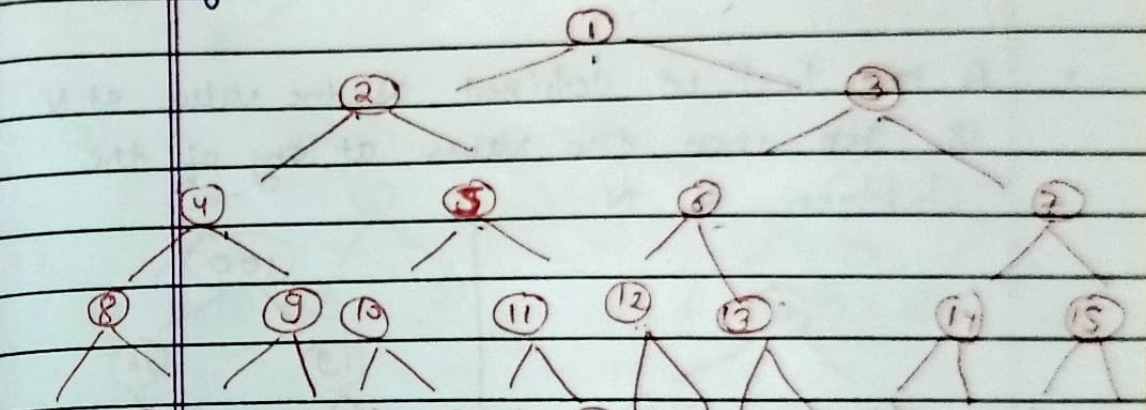


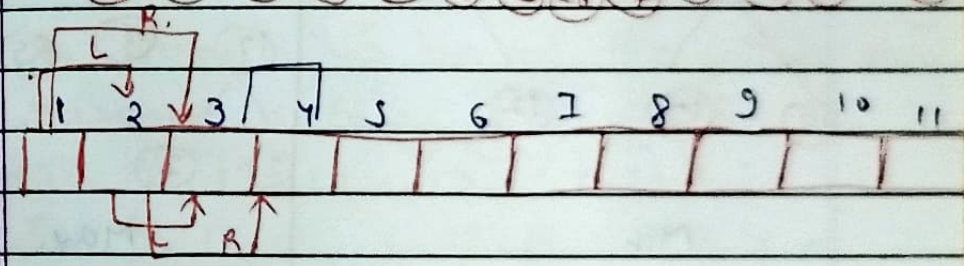
that is not Complete binary tree.

* It is consider as array.

* One can easily determine the children and Parent of a node K in ~~any~~ any complete tree T.

* Specially the left / Right children of the node K are $2*K$, $2*K+1$ and the Parent of K is the node lower bound $(K/2)$.





22 का Parent = $\frac{22}{2} = (11)$

Left child of K = 2K

Right " " K = 2K+1.

Left child 7 = 14.

Right child of 7 = 14+1 = 15.

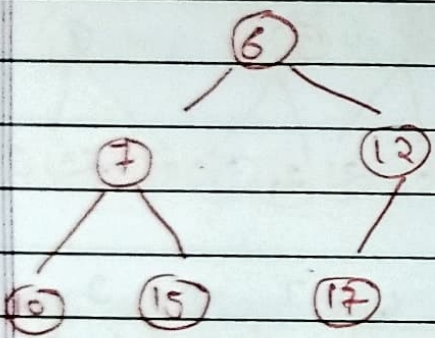
Array Pointer based on की Requirement नहीं होती।

* Heap :-

Suppose H is a Complete binary tree with n element, H is called a Heap, if each node N of H has following Properties:

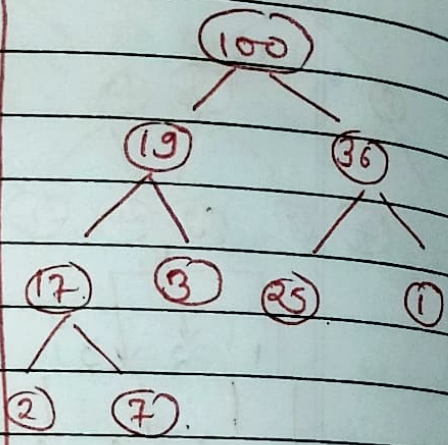
- The value of N is greater than to the value at each of the children of N then it is called Max heap.

- A min heap is defined as the value at N is less then the value at any of the children of N.



Min

[Parent children से छोटा होना चाहिए]

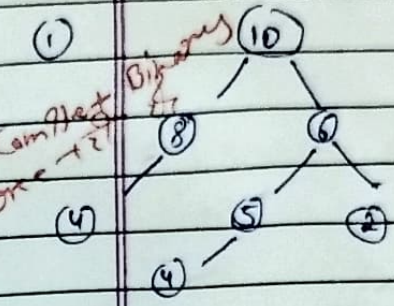


Max.

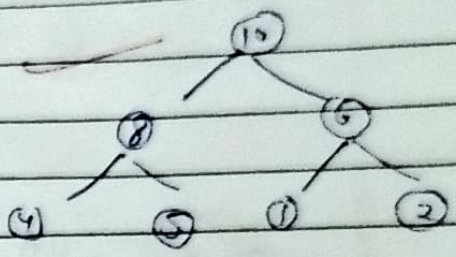
[Parent children से बड़ा होना चाहिए]

Q → A max-heap is a heap where the value of each parent is greater than or equal to the value of its children. Which of the following is a max heap

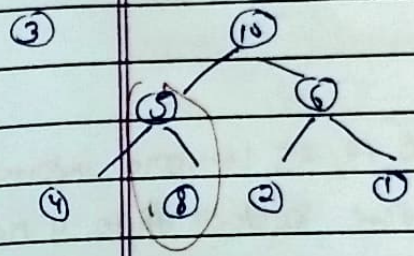
(1) Complete Binary tree



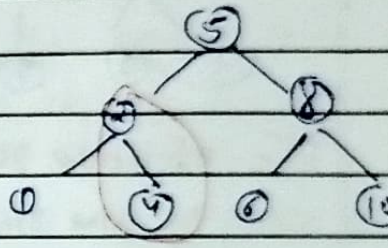
(2)



(3)



(4)



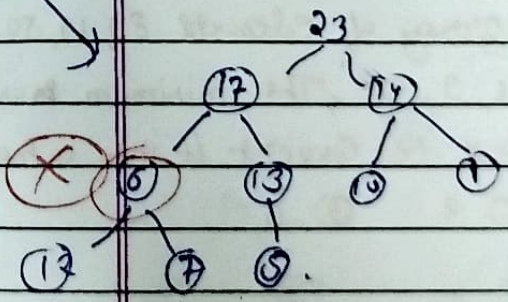
Q → Consider a binary max-heap implemented using an array. Which one of the following array represents a binary max-heap.

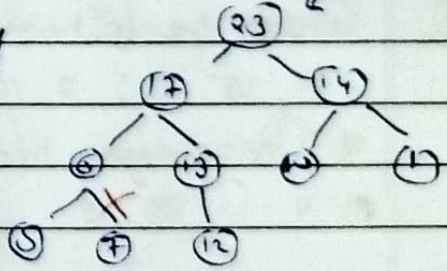
(A) 23, 17, 14, 6, 13, 10, 1, 12, 7, 5.

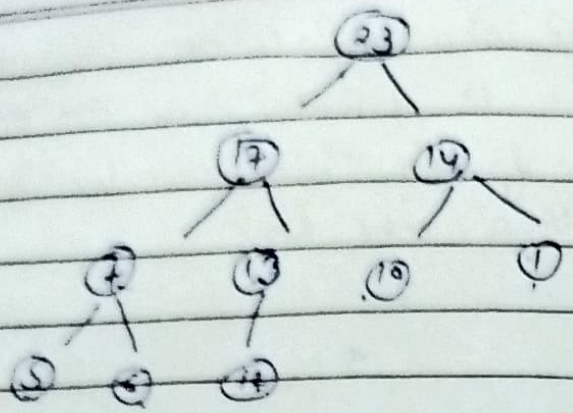
(B) 23, 17, 14, 6, 13, 10, 1, 5, 7, 12

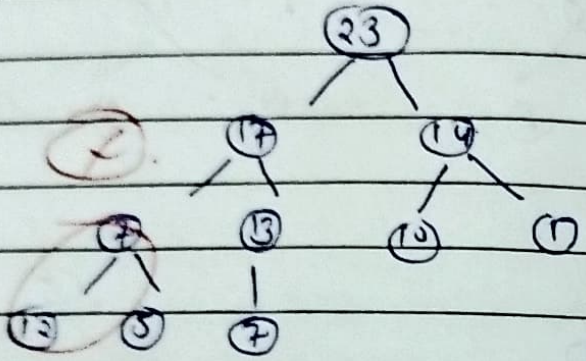
(C) 23, 17, 14, 7, 13, 10, 1, 5, 6, 12.

(D) 23, 17, 14, 7, 13, 10, 1, 12, 5, 7.



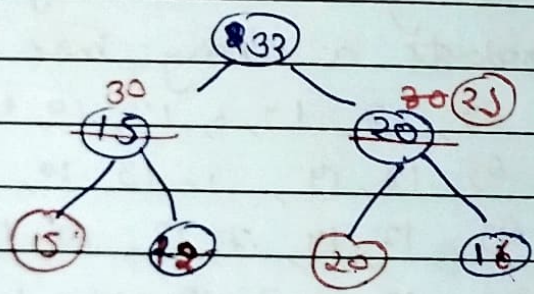






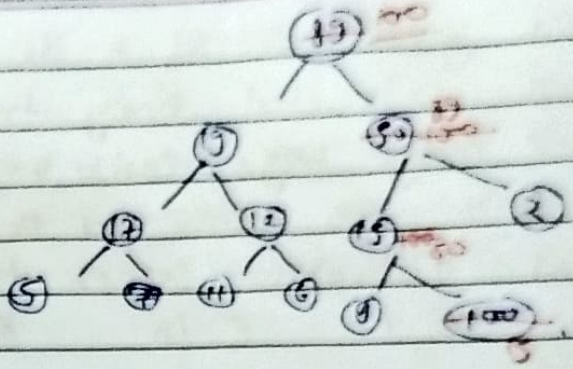
Q → The elements 35, 15, 20, 30, 12, 25, 16 are inserted one by one in the given order into a Max Heap. The resultant Max Heap is.

इस Step पर check करना है Max heap है की नहीं



Q → Consider the following array of elements 89, 19, 50, 17, 12, 15, 2, 5, 7, 11, 6, 9, 100. The minimum number of interchanges needed to convert it into a max-heap

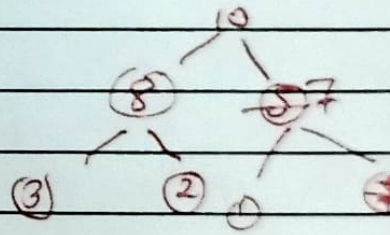
(A) 4 (B) 5 (C) 2 (D) 3.



Q A Priority queue is implemented as a Max heap initially. It has 5 elements. The level order traversal of the heap is 10, 8, 5, 3, 2. Two new elements 1 and 7 are inserted into the heap in that order. The level-order traversal of the heap after the insertion of the elements is

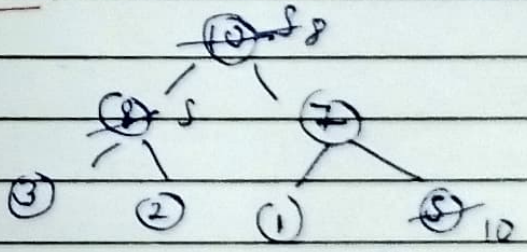
(a) 10, 8, 7, 3, 2, 1, 5 (b) 10, 8, 7, 2, 3, 1, 5

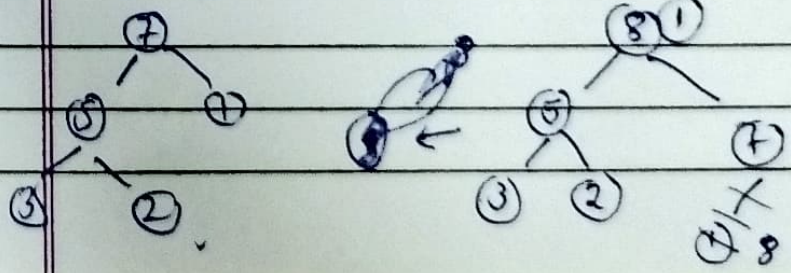
(c) 10, 8, 7, 1, 2, 3, 5 (d) 10, 8, 7, 5, 3, 2, 1



10 8 7 3 2 1 5

Delete in heap ←





6/8/10:28

Date
Page

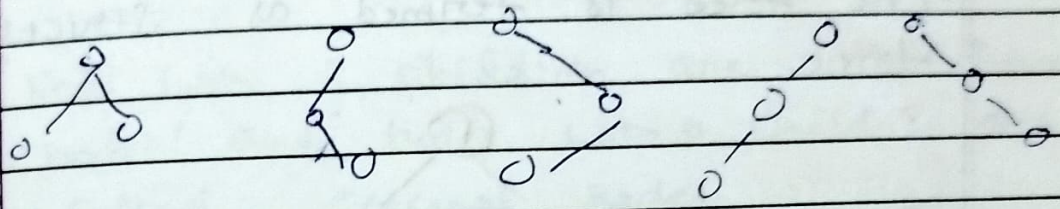
X We are given a set of n distinct elements and an unlabeled binary tree with n nodes. In how many ways can we populate the tree with the given set so that it becomes a binary search tree?

(a) 0 (b) 1 (c) 6 (d) $\frac{1}{(n+1)} {}^{2n}C_n$

$$\frac{{}^{2n}C_n}{n+1} =$$

Q The maximum num of binary tree that can be formed with three unlabelled node is: —

(a) 1 (b) 5 (c) 4 (d) 3



6:B,
Q How many distinct binary search tree can be created out of 4 distinct keys?

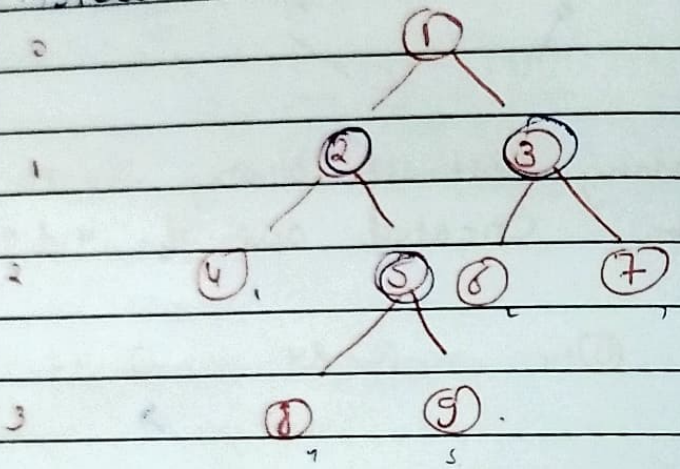
(A) 4 (B) 14 (C) 24 (D) 42

Ex2

Data
Page

Strictly binary tree

If every non-leaf node in a binary tree has non empty left or right subtree the tree is termed as Strictly binary tree.



A Strictly binary tree with n leaves always contain 2n-1 nodes.

"If every non leaf node in a binary tree has exactly two children, the tree is know as Strictly binary tree."

$2 \times 3 - 1 = 6 - 1 = 5$

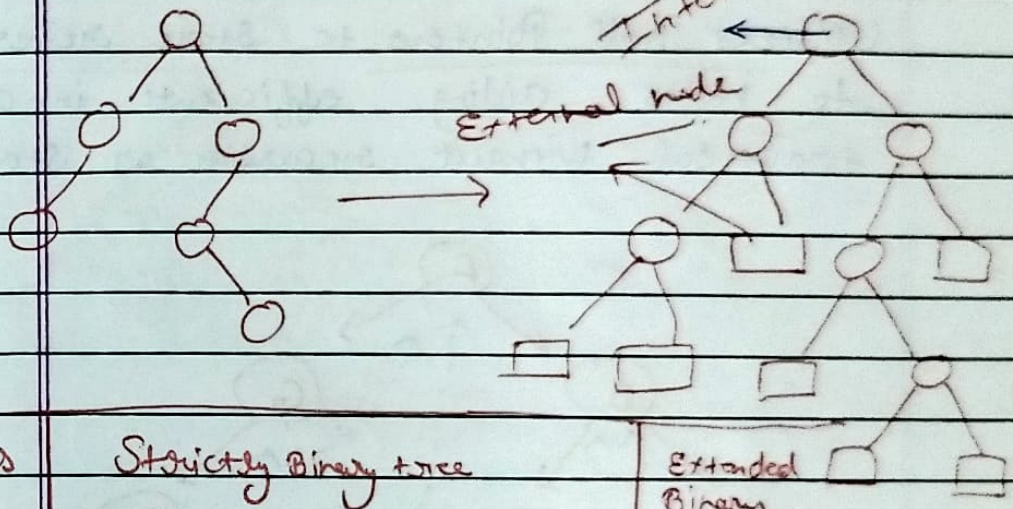
6:15

Date
Page

Extended binary tree / 2-tree

if each node has either 0 or 2 children.

Node with 2 children are called internal nodes and nodes with 0 children are called external nodes.



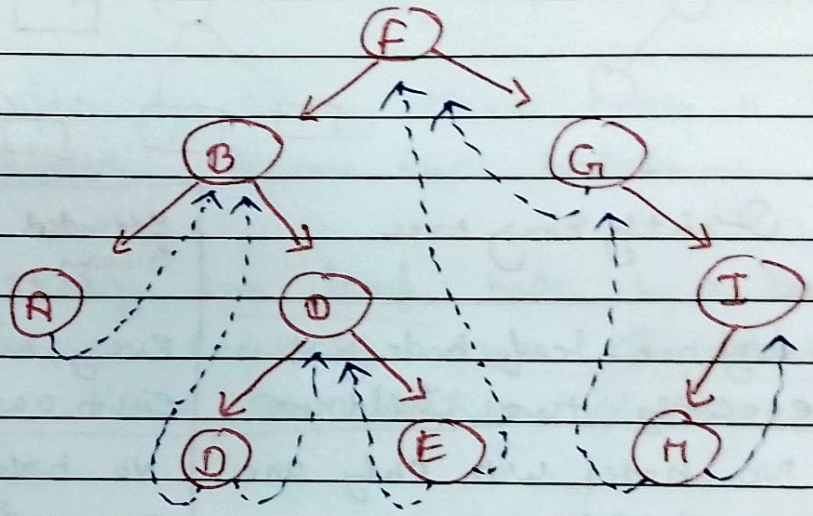
features	Strictly Binary tree	Extended Binary tree
Definition	Every non-leaf node has exactly two children.	Every node has either 0 or 2 children.
Node with one children	No nodes with only one child.	No node with only one child.
Special node	None.	May have external (dummy) nodes to make it full.
Typical usage	Less common in Practical application.	Used in Science like Huffman coding trees where it's beneficial to consider the tree as full.

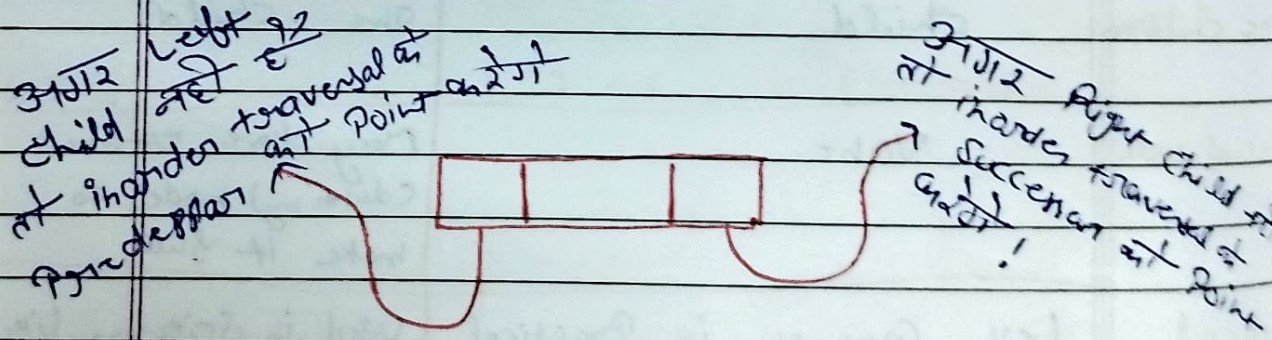
(x) Threaded binary tree

- A threaded binary tree is a modified binary tree that uses null Pointers to link to the next node in an in-order sequence, optimizing in-order traversal.

Purpose: →

Utilizes null Pointers to store references to nodes, aiding efficient in-order traversal without recursion or stacks.

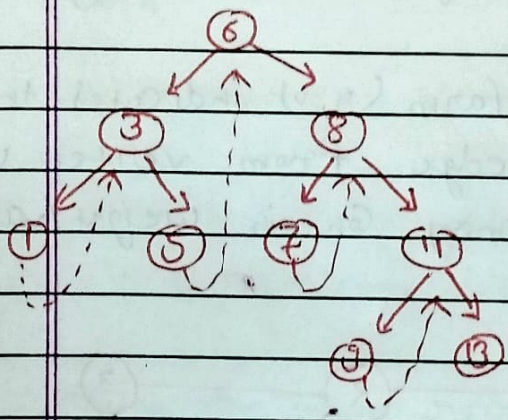


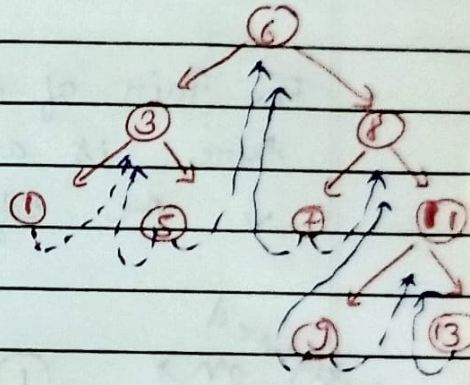


Types :—

- Single Threaded :— Nodes threaded towards either inorder Predecessor / Successor.

- Double Threaded :— Nodes threaded towards both Predecessor and Successor.





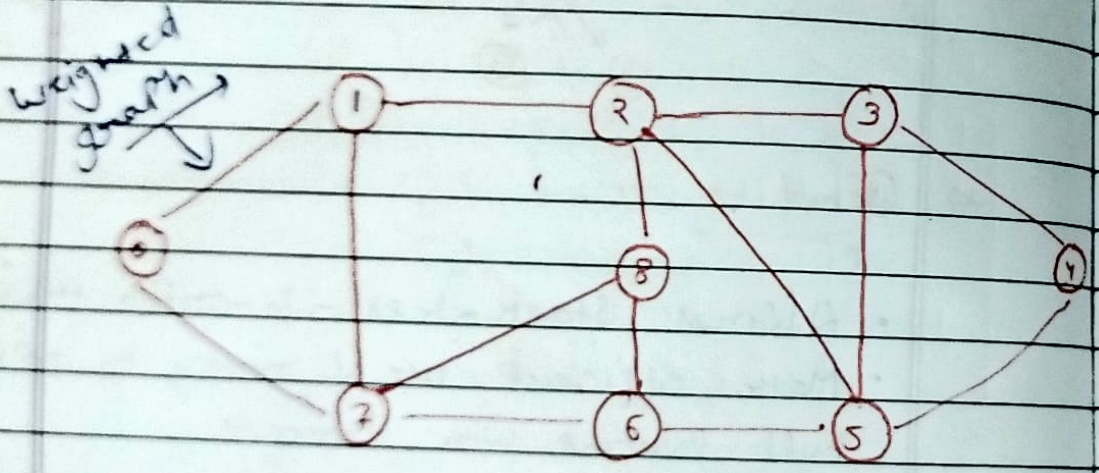
* Benefits :—

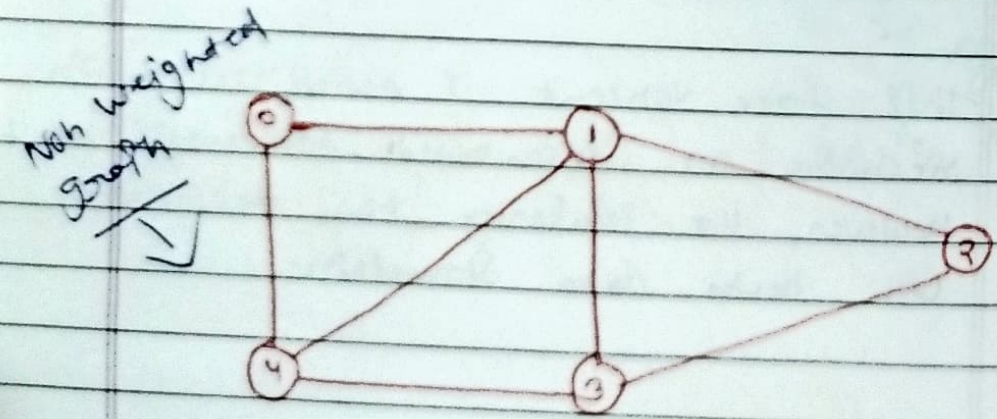
- Allows Stack-less - in-order traversal
- Make efficient use of memory by replacing null pointers with threads.

- This tree varient is beneficial when recursive or Stack-based traversals aren't. However, its popularity has decreased with newer data structures.

Graph

- Graph is the data structure that consists of following two components.
- A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of a directed graph (di-graph).
- The pair of the form (u, v) indicated that there is an edge from vertex u to v. The edge may contain weight/value/cost.





Graph are used to represent many real life application. Graph are used to represent network. The network may include path in a city or telephone network.

Graph are also used in social network like linkedin, facebook, in facebook each person is represented with a vertex (node) each node is a structure and contain information like person Id, name, gender.

Representation of graph is many

* Adjacency Matrix
* Adjecency List.

There are other representations also like Incidence Matrix and incidence List. The choice of the graph representation is situation specific. It totally depends the type of operations to be performed and ease of use.

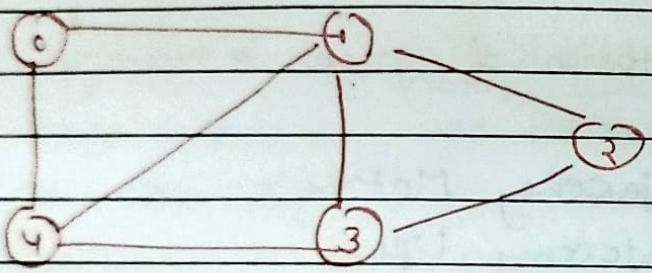
Adjacency Matrix

Adjacency Matrix is a 2D array of size V x V where V is the num of vertex in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates

that there is an edge from vertex i to vertex j.

Adjacency matrix for undirected graph is always symmetric.

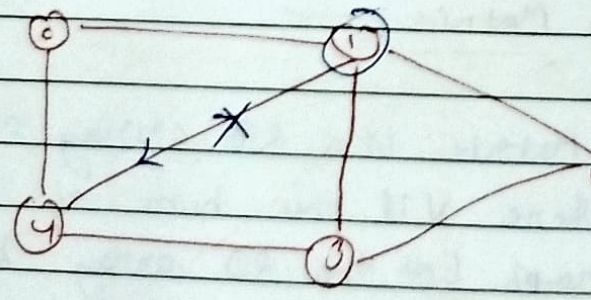
Adjacency Matrix is also used to represent weighted graphs, if adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.



	0	1	2	3	4
0					1
1	1		1	1	1
2		1		1	
3		1	1		1
4	1	1		1	

S+S.

for directed graph.



	0	1	2	3	4
0					1
1				1	
2					
3					
4	X				

Incidence Matrix

Representation of undirected graph

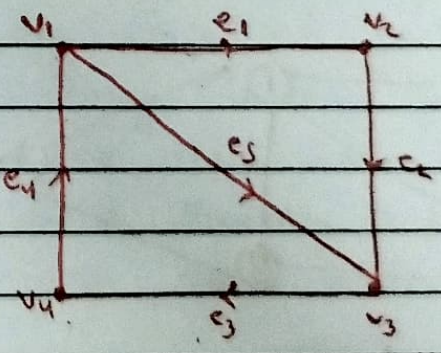
Consider a undirected graph G = (V, E) which has n vertices and m edges all labelled. The incidence matrix I(G) = [bi,j], is then n × m matrix.

- Where bi,j = 1 when edge ej is incident with vi
  = 0 otherwise.

Representation of directed graph

The incidence matrix I[D] = [bi,j] of digraph D with n vertices and m edges is the n × m matrix in which

- Bi,j = 1 if arc j is directed away from vertex vi
  = -1 if arc j is directed towards vertex vi
  = 0 otherwise



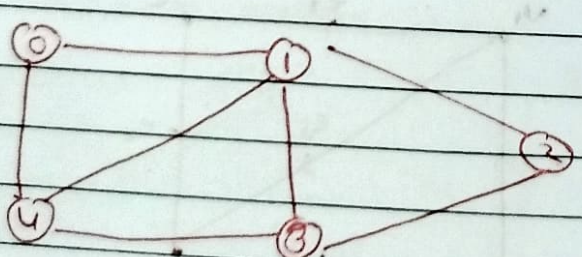
The matrix of the graph is:-

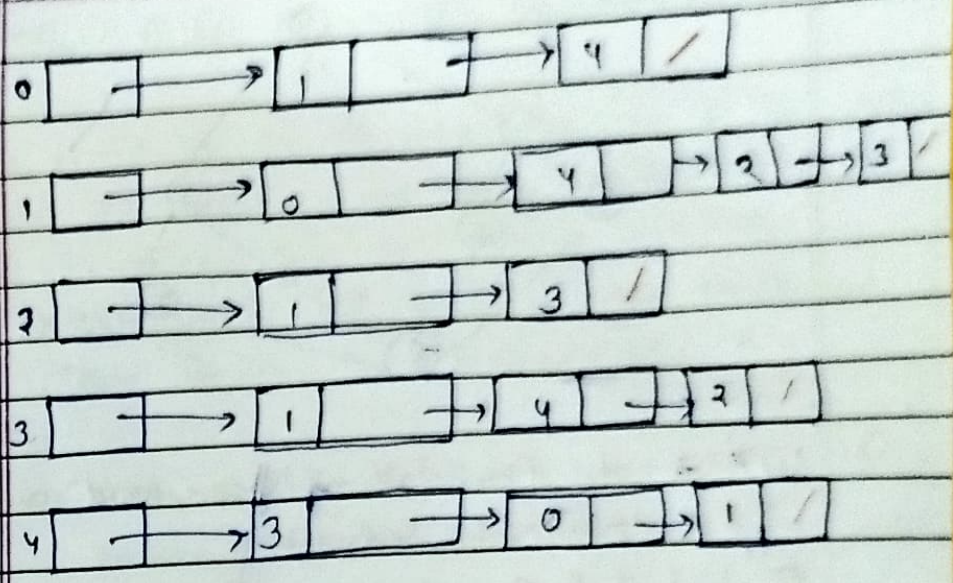
$$I(D) = \begin{array}{c|ccccc} & e_1 & e_2 & e_3 & e_4 & e_5 \\ \hline v_1 & 1 & 0 & 0 & -1 & 1 \\ v_2 & -1 & 1 & 0 & 0 & 0 \\ v_3 & 0 & -1 & 1 & 0 & -1 \\ v_4 & 0 & 0 & -1 & 1 & 0 \end{array}$$

$i \longrightarrow j$

	e
i	1
j	-1

Adjacency List: — An array of list is used. Size of array is equal to the num of vertices. Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as list of pairs.





a. for non-weighted graph | INFO | Adj-Lists |

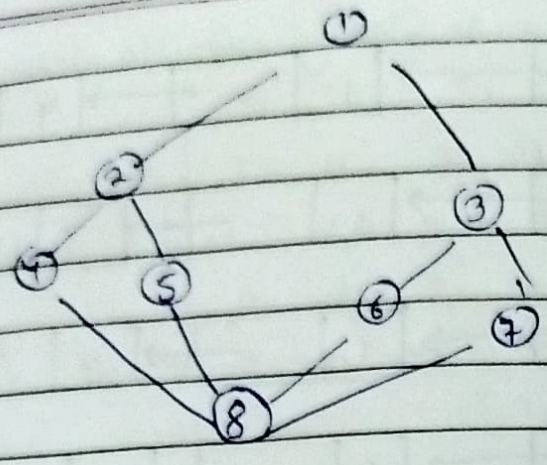
b. for weighted graph | Weight | INFO | Adj-Lists |

Graph Traversal :

- Traversal means visiting all the nodes of a graph.
- Depth first Traversal (or Search) for a graph is similar to Depth first traversal of a tree.
- The only catch here is, Unlike trees, graph may contain cycles. So we may come to the same node again. To avoid processing a node more than once, we use a Boolean visited array.

BFS →
DFS →

Backtrack



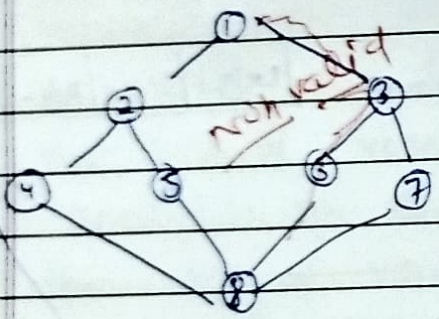
Q Which of the following are valid and invalid DFS traversal sequence

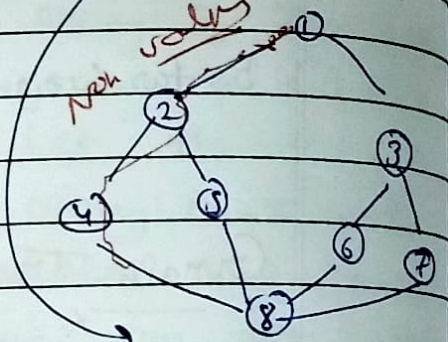
(a) 1, 3, 7, 8, 5, 2, 4, 6

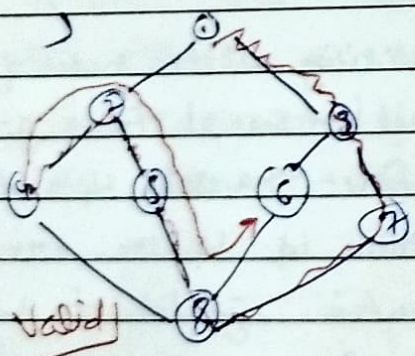
(b) 1, 2, 5, 8, 6, 3, 7, 4.

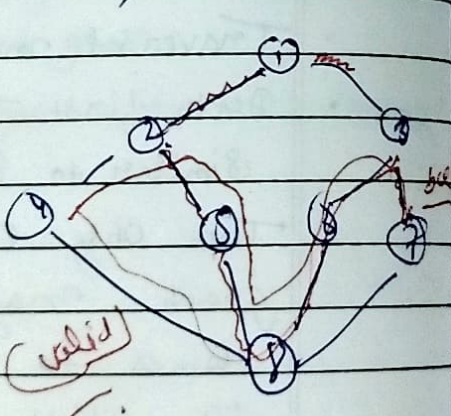
(c) 1, 3, 8, 7, 2, 5, 2, 4

(d) 1, 2, 4, 5, 8, 6, 7, 3.









The DFS algorithm works as follows.

- Start by putting any one of the graph's vertices on top of a stack.
- Take the top item of the stack and add it to the visited list.
- Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of stack
- Keep repeating steps 2 and 3 until the stack is empty.

DFS(v)

```
{
    visited(v) = 1
    for all x adjacent to v
    {
        if (x is not visited)
            DFS(x)
    }
}
```

Importance of DFS :-

- Testing whether graph is connected.
- Computing a spanning forest of G.
- Computing the connected components of G.
- Computing a path b/w two vertices of G or reporting that no such path exist.
- Computing a cycle in G or reporting that no such cycle exists.

Application :-

- finding connected components.
- Topological sorting.
- finding 2-(edge or vertex)-connected components.
- " 3- " " " "
- find bridges of graph.
- finding strongly connected component.

BFS :- Breadth for search :-

- Breadth first search for a graph is similar to ~~binary~~ DFS. The only catch here is unlike trees, graph may contain cycle. So we may come to the same node again.

To avoid processing a node more than once we use a boolean visited array similarly

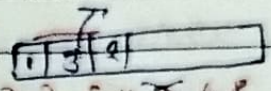
BFS → Checking by children nearby.

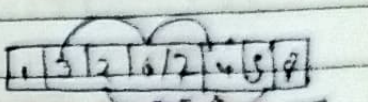
- 30:4

It is assumed that all vertices are reachable from the starting vertex i.e the graph is connected

Just insert at 3rd child first

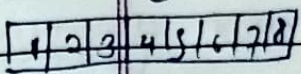
→ BFS

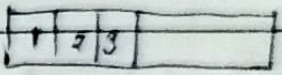


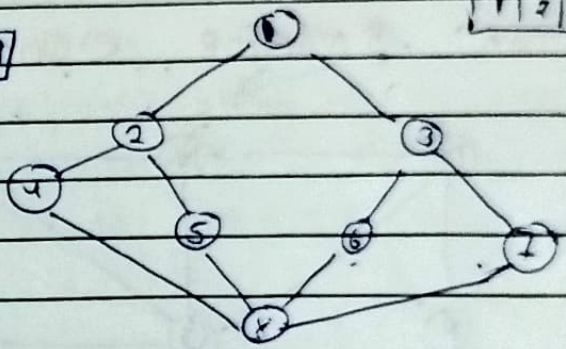


① 1, 3, 2, 5, 4, 7, 6, 8

② 1, 3, 2, 5, 4, 7, 6, 8







BSf (v)
{

visited (v) = 1
insert [v, Q]
while (Q! = Φ)
{

u = Delete (Q);

for all n adjacent to u
{

if (n is not visited)
{

visited [n] = 1
insert (n, Q)

}

}

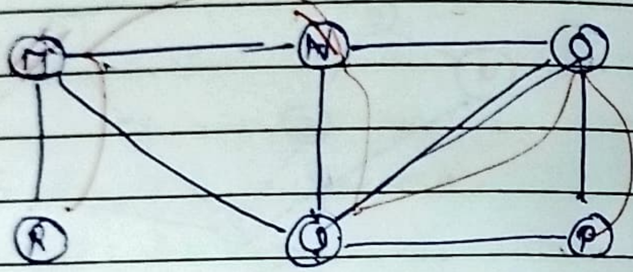
}

time complexity $O(|V| + |E|)$.

23rd

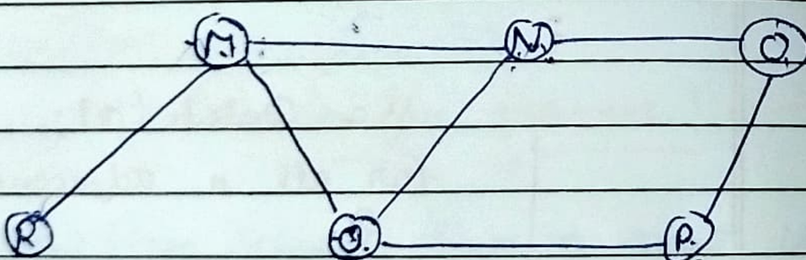
Q. BFS has been implemented using queue data structure which one of the following is a possible order of visiting the node in the graph above

(a) MNOPQR (b) NQMPOR (c) QMNROP (d) POQNMR



(a) MNOPQR (b) NQMPOR (c) QMNPRO

(d) QMNPOR



Importance of BFS :-

* One of the single source shortest Path algorithm, so it is used to compute the shortest Path.

* It is also used to solve Puzzles such as the Rubik's Cube.

* BFS is not only the quickest way of solving the Rubik's Cube, but also the most optimal way of solving it.

Application :-

- Copying garbage Collection
- finding the shortest Path b/w node u and v with Path length measured by number of edges.
- Ford-Fulkerson method for Computing the maximum flow in a flow network
- Serialization/Deserialization of binary tree vs serialization in sorted order, allow the tree to be reconstructed in the efficient manner.
- Testing bipartiteness of a graph.

hashing ... • But time ... search ... element ...

Hashing :—

Main idea of data structure is to help us store the data. But most common operation on any data structure is not insert / delete but actually search, as even for insertion / deletion search is also required.

In any of the data structure the search time first depends on the number of elements which data structure contains and then on type of structure.

g —

Unsorted array - O(n)
Sorted array - O(log n)
Link list - O(n)
BT - O(n)
BST - O(n)
AVL - O (log n).

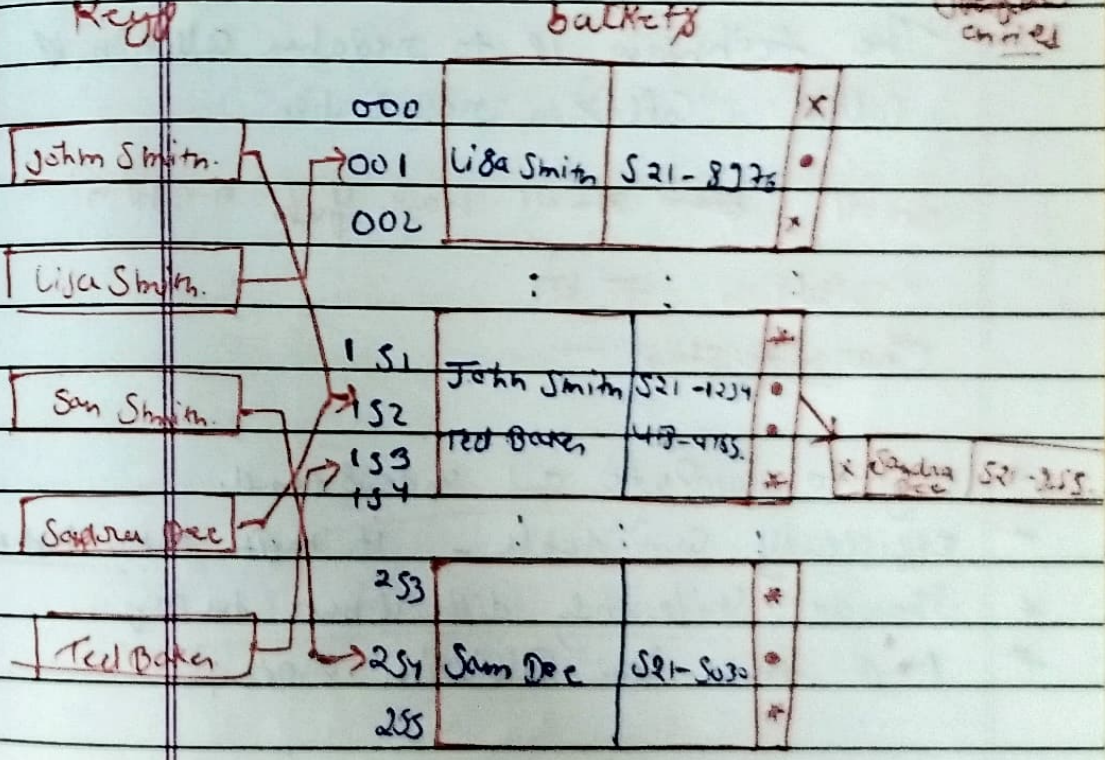
Key का use करके एक location generate कर लेते है

12 | 12 | 2012 = 1+2+1+2+2+0+1+2 = 11
11 = 1+1 = 2

→ So hashing is a technique where search time is independent of the num of items in which we are searching a data value.

→ The basic idea is to use the key value to find the address in the memory to make searching easy. For eg to the phone num, we use Adhar Card / Voter Id or any other key and convert it into a smaller practical num as index in a table called hash table.

The values are then stored in hash table. By using that key you can access the element in O(1) time.



Q: 8

Q. Given the following input (4322, 1334, 1471, 9679, 1989, 6171, 6173, 4199) and the hash function x mod 10, which of the following statements are true:

(i) 9679, 1989, 4199 hash to the same (i)

(ii) 1471, 6171 hash to the same value

(iii) All elements hash to the same value ✗

(iv) Each element hash to the diff value ✗

Collision — It is possible that two diff set of keys K_1 and K_2 will yield the same hash address. This situation is called Collision.

The technique is to resolve collision is called collision resolution.

ऐसा अच्छा hash fun बनाओ की collision न हो

Characteristics :—

- Easy to compute and understand.
- Efficiently computable, it must take less time.
- Should uniformly distribute the key.
- Must have low collision rate.

Most Popular hash function :—

Division-remainder method :—

The size of the num of items in the table is estimated. That num is then used as a divisor into each original value as key to extract a quotient and remainder.

The remainder is the hashed value. (Since this method is liable to produce a num of collisions, any search mechanism would have to be able to recognize a collision and offer an alternate search mechanism.

$$H(K) = K \pmod m$$
$$H(K) = K \pmod m + 1$$

Mid Square method.

folding method.

e to Amplify.

Collision Resolution Techniques :—

Open Addressing / closed hashing :—

All elements are stored in the hash table itself. ie collision is resolved by Probing searching through alternate location within it self in a particular sequence

When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table. So at any point, size of table must be greater than or equal to total number of keys.

3 type of Probing :—

* linear
* quadratic
* double

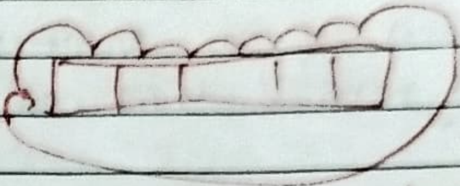
Linear Probing :—

In linear Probing method, in case of a collision we find out the next free slot and store the key that is causing collision it in.

Method —

$h(k, i) = (h'(k) + i) \bmod m;$

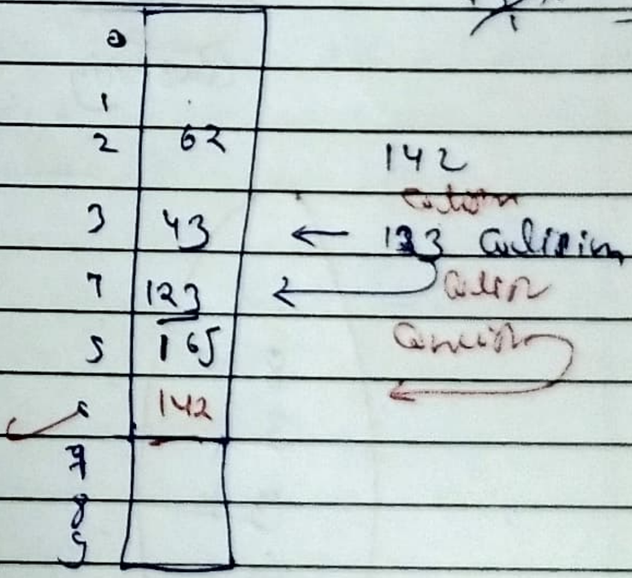
for $i = 0, 1, \ldots, m-1$.



Q) A hash table contains 10 buckets and uses linear probing to resolve collision. The key values are integers and the hash function used is key % 10. If the values 43, 165, 62, 123, 142 are inserted in the table, in what location would the key value 142 be inserted.

(a) 2 (b) 3 (c) 4 (d) 6

43 % 10 = 3



इसमें clustering की Problem होने लगेगी सारा data एक साथ इकट्ठा होने लगेगा।

instead of

Date
Page

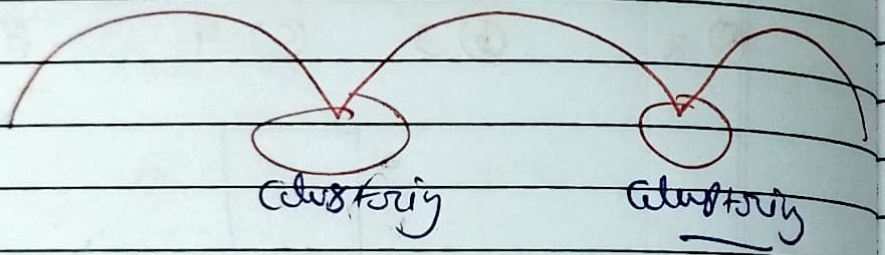
Quadric Probing

Quadric Probing operation by taking the original hash index and adding successive values of an arbitary quadratic Polynomial until an open slot is found.

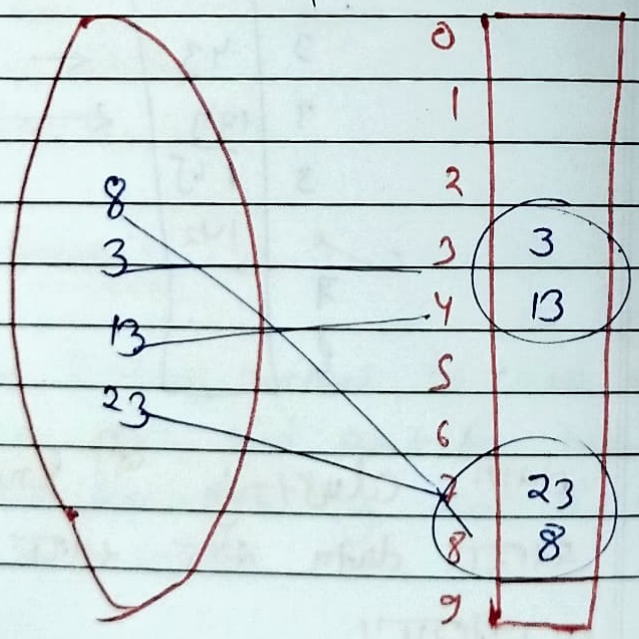
Quadratic Probing uses a hash function of the form.

$$h(k,i) = (h'(k) + f(i^2)) \bmod m.$$

Where, h' is an auxiliary hash function and $i = 0, 1, \ldots, m-1$.



Eg:



8, 3, 13, 23

$h(8) = [(h(8) + f(0)] \bmod 10 = 8$ so it get location 8.

3 will be Placed at $h(3) = [h(3) + f(0)] \bmod 10 = 3$ no collision so it is " " " 3

13 " " " " " $h(13) = [h(13) + f(0)] \bmod 10 = 3$ collision occurs so we increase the value of i.

$h(13) = [h(8) + f(1)] \bmod 10 = 4$, no collision so it " " " 4. "

23 " " " " " $h(23) = [h(23) + f(0)]$ mod 10 = 3 collision occurs increase the value of i).

$h(23) = [h(23) + f(2)] \bmod 10 =$

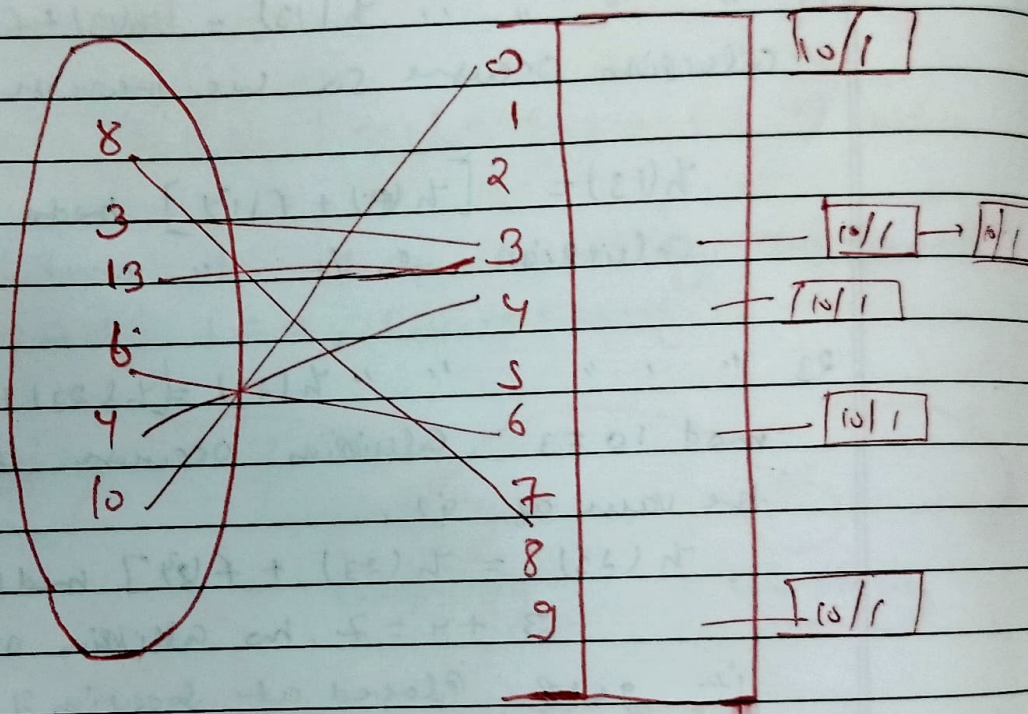
3 + 4 = 7, no collision occurred so it ~~got~~ Placed at location 7.

Quadratic Probing avoid clustering of elements and this improve the search time.

Chaining :—

→

The idea is to make each cell of hash table point to a linked list of records that have same hash function value in Chaining, we place all the elements that hash to the same slot into the same linked list.



Double Hashing :— hashing करने के
लिए दो hash functions का use
hashing करने के to resolve collision.