Hashing.

Types of Hashing

① Open Hashing (closed Addressing) — size of the table is unlimited
② Closed Hashing (Open Addressing) — size is limited

- Linear probing
- Quadratic Probing
- Double Hashing

Chaining

* Chaining method to resolve the collision:-

Eg. A = 3, 2, 9, 6, 11, 13, 7, 12

$h(K) = 2K + 3$ $m = 10$.

Use division method and closed addressing to store these values.

$h(K) = 2K+3 \Rightarrow$ hash function

Key	Location ($K \% m$)
3	$[(2\times3)+3] \% 10 = 9$
2	$[(2\times2)+3] \% 10 = 7$
9	$[(2\times9)+3] \% 10 = 1$
6	$[(2\times6)+9] \% 10 = 5$
11	$[(2\times11)+3] \% 10 = 5$
13	$[(2\times13)+3] \% 10 = 9$
7	$[(2\times7)+3] \% 10 = 7$
12	$[(2\times12)+3] \% 10 = 7$

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

Open addressing ⇒ by default use Linear probing.

Eg. Linear probing :- (means finding / searching)

Use Division method and open addressing to store these values.

A = 3, 2, 9, 6, 11, 13, 9, 12.

h(K) = 2K + 3.

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

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

No. of times we search free space and put the element in the hash table.

1, 2, 4, 9, 12.

Order of elements in hash table:

⇒ 13, 9, 12, –, –, 6, 11, 2, 7, 3.

In simple words: Insert K_i at first free location from $(u+i)\%m$ where $i = 0$ to $(m-1)$

(m is size of hashtable)

* Quadratic Probing: Searching is done in such a manner that —

Disadvantages of linear probing → there is a problem of primary clustering.

i = 0
i = 1
i = 4
i = 9
i = 16.

} In quadratic i.e $i = i \times i$ manner.

Eg. 4371, 1323, 6173, 4199, 4344, 9679, 1989

	Flag table		Key	Location (u)	Probes
0	9679	1			
1	4371	1			
2		0	4371	4371 % 10 = 1	1
3	1323	1	1323	1323 % 10 = 3	1
4	6173	1	6173	6173 % 10 = 3	
5	4344	0			
6	~~1989~~	1			
7	~~6173~~	1			
8	1989	0			
9	4199	1			

Note: hash loc^n is zeroth

initially all are zero.

Formula: loc = loc + (i×i) % 10. where i = 0, 1, 2, 3 ...

Code to insert elements through quadratic probing.

```
void insert(int key)
{
    loc = key % 10;
```

Disadvantage ⇒ Although there would be free spaces still they won't be mapped (Quadratic)

```
for (int i = 0; i < 10; i++)
    {
        loc = loc + (i * i) % 10
        if (flag [loc] == 0)
            { hashtable [loc] = key;
              flag[loc] = 1;
              break;
            }
        else
            {
```

* Double Hashing:

Formula: f ⇒ hash fun^n.

$f_1(key)$ → i, where i = 1 to n.

⇒ $f_1(key) + 1 \times f_2(key)$ ← if occupied

⇒ $f_1(key) + 2 \times f_2(key)$ ← if occupied

⇒ $f_1(key) + 3 \times f_2(key)$ ← if occupied

Eg. Formula: ~~hash fun^n~~

4371, 1323, 6173, 4199, 4344, 9679, 1989.

f_1 = key % 10. ✓

f_2 = 7 − (x % 7) ✓

→ key.

→ ① 4371 % 10 = 1

② 1323 % 10 = 3

0	
1	4371
2	
3	1323
4	6173
5	9679
6	
7	4344
8	
9	4199

520
7)4344
42
14
14
4

③ 6173 % 10 = 3

④ Using formula

$f_1(key) + 1 \times f_2(key)$

$3 + 1(7 - (x \% 7))$

$3 + 1(7 - 6)$

$3 + 1$

$= 4$ ✓

∴ Insert ~~at~~ 6173 at (4) location

④ 4199 % 10 = 9.

⑤ 4344 % 10 = 4

$4 + 1(7 - (x \% 7))$

$4 + 1(7 - 4)$

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

⑥ 9679 % 10 = 9.

$9 + 1($

283
~~13~~)1989
14
58
56
29
7

⑦ 1989 % 10 = 9

$9 + 1(7 - (x \% 7))$

$9 + 1(7 - 7)$

$9 + 1(0) = 9$

Now i = 2.

$9 + 2[7 - (0)$

∴ Even though cells are empty, still we cannot insert 1989. And hence this is the disadvantage of double hashing.

9/10

Eg 22, 9, 5, 18, 14, 28, 30, 19.

f_1 = Key % 10

f_2 = 1 + key % 9.

0	10
1	19
2	22
3	
4	14
5	5
6	30
7	
8	18
9	9

① 22 % 10 = 2

② 9 % 10 = 9

③ 5 % 10 = 5

0 + 1 + 5 % 9

0 + 1 + 0

1

④ 18 % 10 = 8.

8 + 1 + 18 % 9

8 + 1 + 0

9

⑤ 14 % 10 = 4

4 + 1 + 14 % 9

4 + 1 + 5

10

⑥ 28 % 10 = 8.

8 + 1 + 28 % 9

8 + 1 + 1

= 10 % 10 = 0

⑦ 30 % 10 = 0

f_2 = 1 + (30 % 9) = 1 + 3 = 4.

0 + 1 + (30 %

0 + 4

Now,

4 + [1 + (30 % 9)]

4 + [1 + 3]

4 + 4 = 8

Now, 8 + [1 + (30 % 9)]

8 + [1 + 3]

8 + 4 = 12 ⟹ 12 % 10 = 2.

Now

2 + [1 + (30 % 9)]

2 + 1 + 3

6

⑧ 19 % 10 = 9

9 + [1 + (19 % 9)]

9 + 1 + 1

11 % 10 = 1.

Code:

```
int insert(int table[], int key, int flag[])
{
    int i,j;
    loc = f1(key) % max;
    incre = f2(key)
    for(i=0 to max)
    {
        j = (loc + i * inc) % max;
        if(flag[j] == 0)
        {
            table[j] = key;
            flag[j] = 1;
            return j;
        }
    }
    return(-1);
```

Note: 7%10 = 7

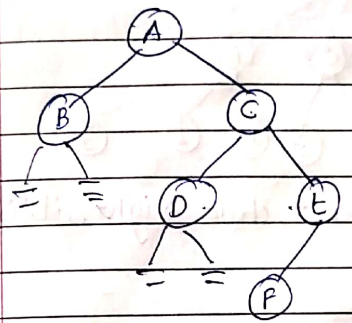
→ Rehashing:

Load factor = $\frac{\text{No. of keys}}{\text{Size of table}}$.

* Extendible Hashing:
→ uses a directory to access its buckets.

DSA

Unit II: Trees.

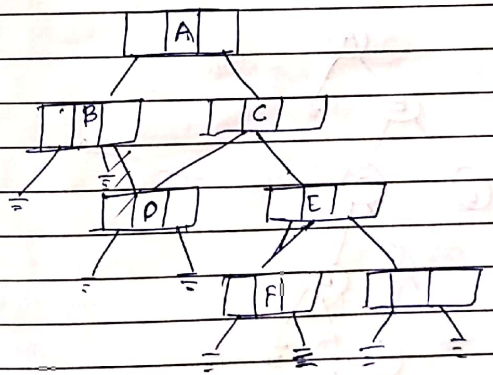


Array Representation of above tree:

No. of nodes

6	A	B	C	\0	\0	D	E	\0	\0	\0	\0	\0	\0	F

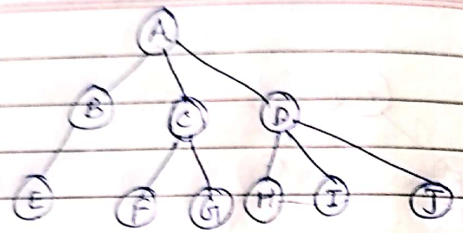
Linked Representation:



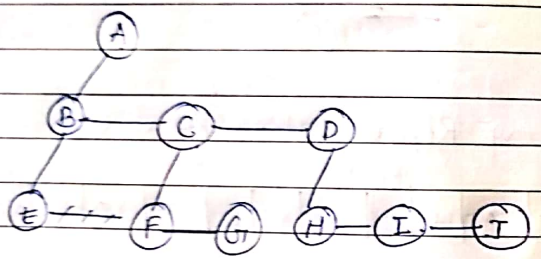
General tree:
→ Node may have more than 2 children.

Converting general tree into binary tree.

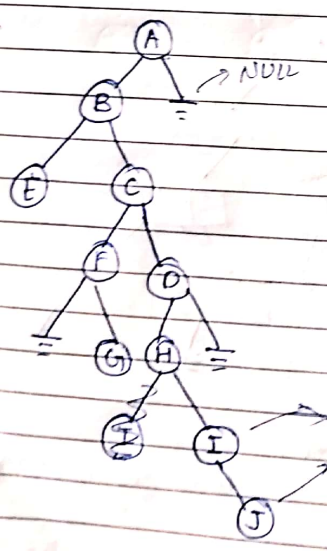
Eg.



Follow: Left most child Right sibling.



Rotate it by 45°.



On right side because I think J and 5 are greater.

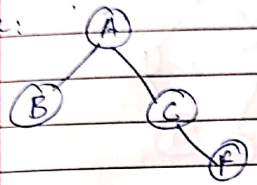
* Construct^{n} of a binary tree

Eg. Inorder: B, I, D, A, G, E, H, F.

Postorder: I D B G C H F E A

Root

Note:



Pre - A B C F

In - B A C F

Post - B F C A

(visiting a node exactly once) Traversal techniques

Depth first

Breadth first

Inorder

Preorder

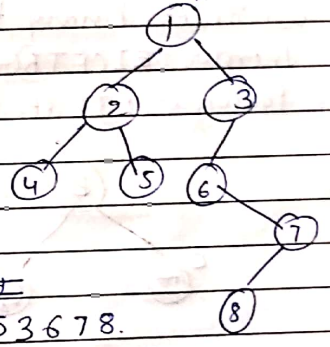
Postorder

Left, right, Root

Levelwise printing of tree.

~~Root, left~~ Left, Root, Right

Root, left, Right



Pre - ~~1 2 3 4~~

1 2 4 5 3 6 7 8.

① Select Root

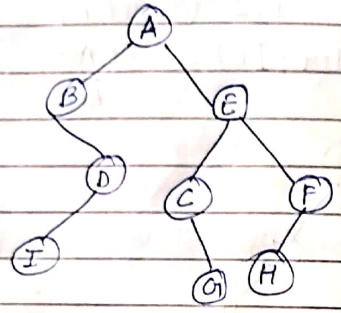
② Traverse left subtree and then Right subtree recursively.

Inorder: 4 2 5 1 6 8 7 3

③ Go to Left most node then write the root node and then go to right, recursively.

Postorder: 4 5 2 8 7 6 3 1

E.g. Soln.

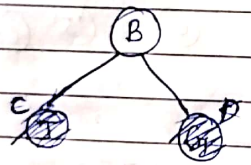


Preorder : A BDIECGFH.

E.g. Construct binary tree:

Inorder : EICFJBGDKHL

Postorder : IEJFCGKLHDB



In : EACKFHDBG

Pre : FAEKCDHGB.

Pre in Post } How to construct

```
class tree
{
    node *root;
  public:
  void create()
  { int x;
    node *p;
    if root is NULL
    root = p;
    p = new node;
    cout << "Enter data
    cin >> x;
    p -> data = x;
    p -> left = NULL;
```

Recursive inorder

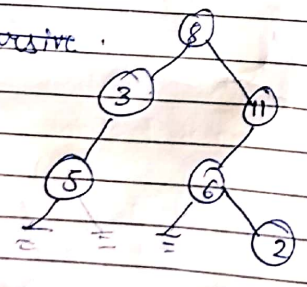
```
node * recinorder (node *p)
{
    if (p != NULL)
    {
        recinorder (p -> left) ;
        cout << p -> data ;
        recinorder (p -> right)
    }
}
```

Recursive preorder:

Same as above:

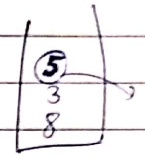
```
{ cout << p -> data ;
    recinorder (p -> left);
    recinorder (p -> right);
}
```

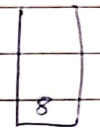
Recursive:

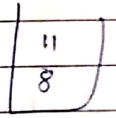


Stack contents

8.	Push 8	
8,3	Push 3	
8,3,5	Push 5	
8,3	Pop 5.	
8,	Pop 3	
8,11	Push 11	Pop 8.
8,11,6	Push 6	Push 11
8,11,	Pop 6.	Push 6
8,	Pop 11	Push 2







Implementation:

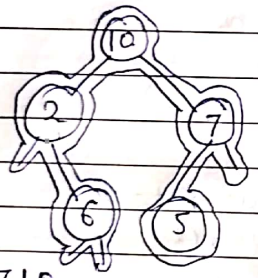
```
while (t != NULL)
{
    cout << t -> data;
    push (t)
    t = t -> left ;
}
if stack is empty
    cout << "
while stack is not empty
{
    t = pop ();
    t = t -> right ;
    while t is not NULL
    {
        display t
        push(t)
    }   t = t -> left
}
```

Code:

```
class Stack
  {
    node *a[10];
    int top;
  public:
    void push(node *t);
```

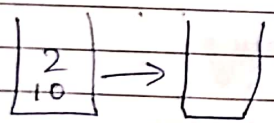
Write post order sequence for the given Binary tree. Write the stepwise contents of the stack.

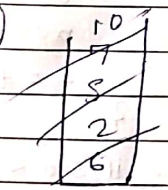
Non-recursive postorder using stack



Left right root

→ 6 2 5 7 10.





0 → left subtree is traversed

1 → Right subtree is traversed.

Pseudo code:

```
while t is not NULL
  {
    push(t, 0)
    t = t->left
  }
while S is not empty
  {
```

BFS → level-wise printing.

↳ Queue is used

```
    t = pop();
    if(t.flag == 0)
      {
        t.flag = 1
        push(t,1)
      }
```

* Breadth First ~~Traversal~~ Search ⇒ level-wise printing.

```
class queue
  {
  private:
    int rear;
    int front;
    int max = 10;
    node * A[max];
    max = 10;
  public:
    queue()
    { rear = front = -1; }
    void isempty()
      {
        if(front == -1 && rear == -1)
          {
            return true;
          }
        else
          {
            return false;
          }
      }
```

insert()

- Recursive algorithm for counting Nodes,

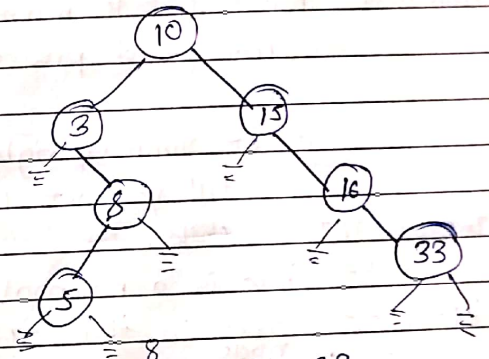
```
int count(node *root)
{
    int i;
    if(root==NULL)
        return 0;
    else
        int i = 1+count(root->left)+count(root->right);
    return(i);
}
```

- Creating exact copy of a tree.

```
node * copy(node *root)
{
    node *p;
    p = NULL;
    if(root != NULL)
    {
        p = new node;
        p->data = root->data;
        p->left = copy(root->left);
        p->right = copy(root->right);
    }
    return(p);
}
```

BST: → Take first element as root node.

10, 15, 3, 8, 16, 33, 5



RLR

Preorder - 10 3 5 8 15 16 33

Inorder - 3 5 8 10 15 16 33 ⇒ Ascending order

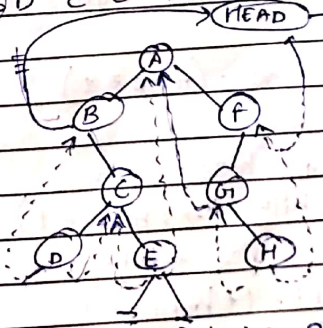
Postorder - 5 8 3 33 16 15 10 → sorted

- Threaded Binary Tree: (TBT)

① Make Inorder Sequence

Replace Left → inorder predecessor
Right → inorder successor.

E.g (B) D C E A G H F



Node Structure
int data
node *l, *r
int lbit, rbit
→ Flags

lbit, rbit ⇒ 0 → thread
lbit, rbit ⇒ 1 → normal link

* Preorder traversal of TBT :-

Pseudocode : if left link is normal
left child preorder successor.

if not climb up right tree
till you get the normal right child.

* Inorder TBT

→ If right child is a normal child then left most node in the ~~right~~ ~~subtree~~ will be the inorder successor.

→ If it is a thread then thread itself is an inorder successor.

Pseudocode :

```
node * inorder (node *t)
{ if (t -> rbit == 1)
    t = t -> right.
  while (t -> lbit == 1)
    t = t -> left.
  return (t)
else
  return (t -> right).
}
void inorder (node * t)
{
  t = head -> left
  while (t -> lbit == 1)
    t = t -> left.
  while (t != head)
  { cout << t -> data;
    t = Inorder (t).
  }
}
```

* Postorder TBT :

Q. Draw TBT equivalent for a given tree assuming root node of a tree starts at index 1.

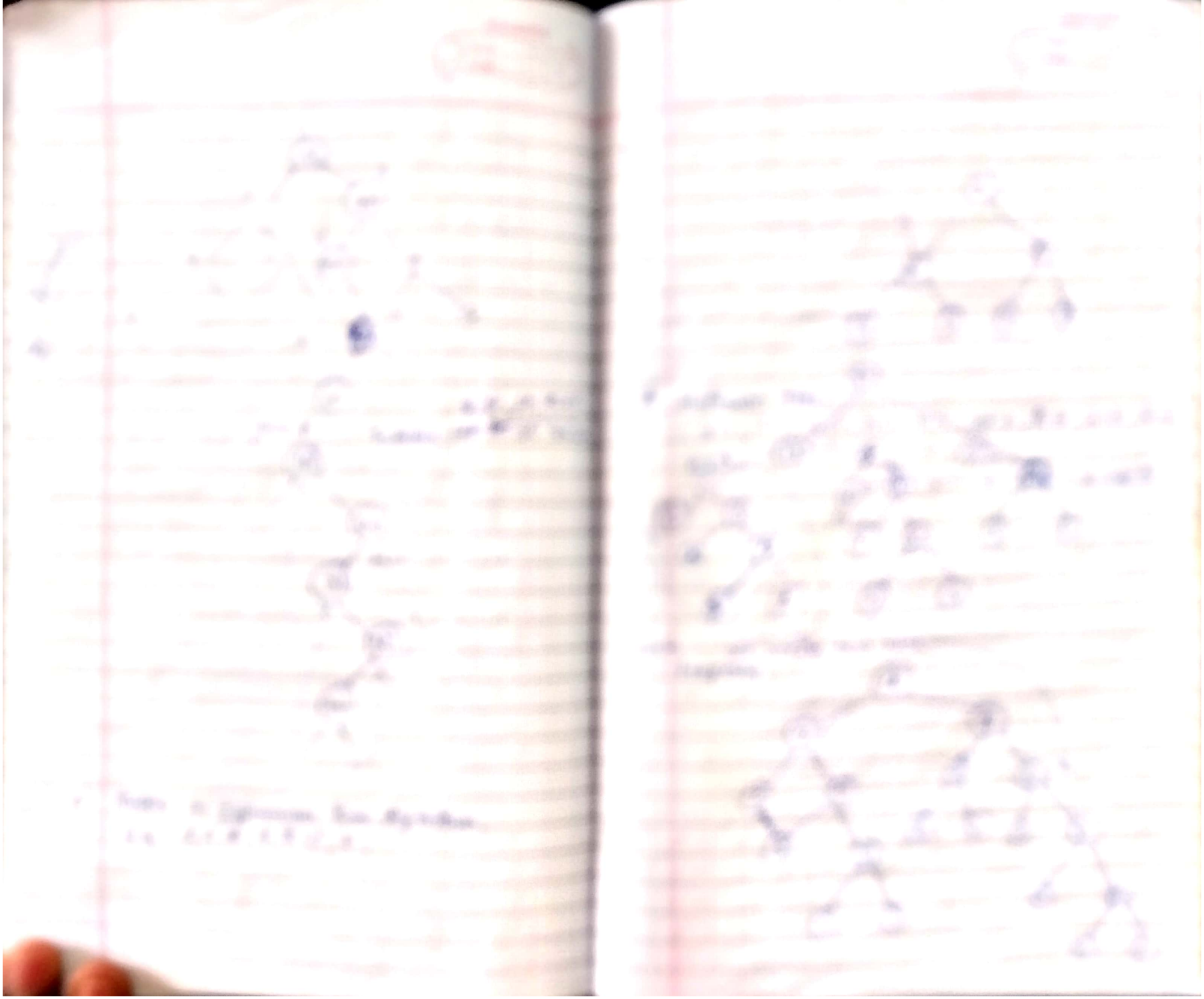
→

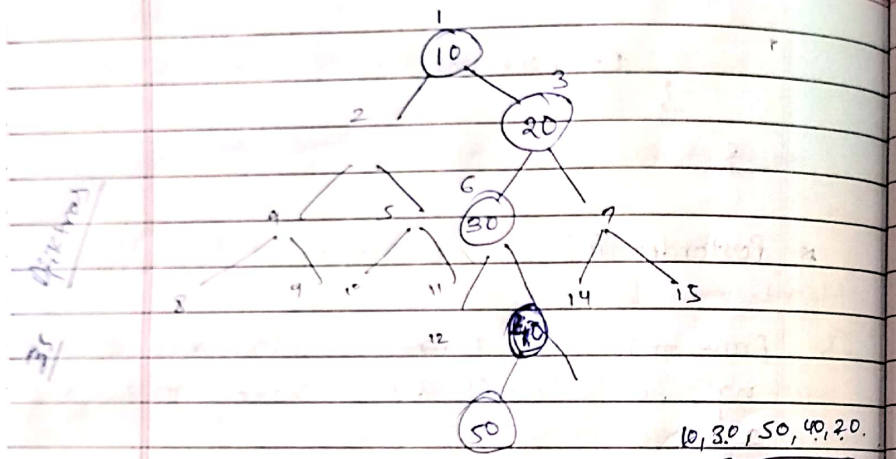
index	1	2	3	4	5	6	7	8
data	—	10	—	20	—	—	30	—	—

9	10	11	12	13	14	15	16	17
—	—	—	—	40	—	—	—	—

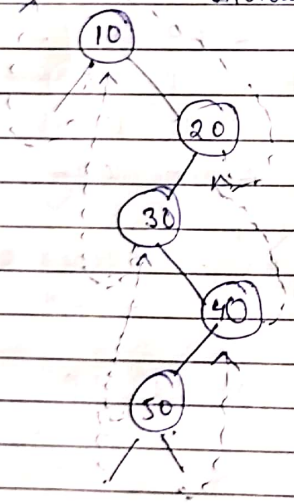
18	19	20	21	22	23	24	25
—	—	—	—	—	—	—	—

26	27	28	29	30	31
50	—	—	—	—	—

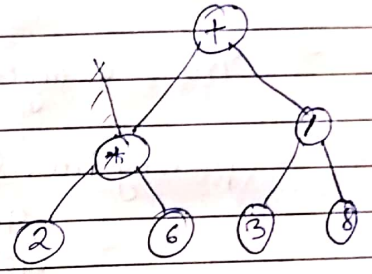




Inorder : ~~10 20~~ 30 40 50

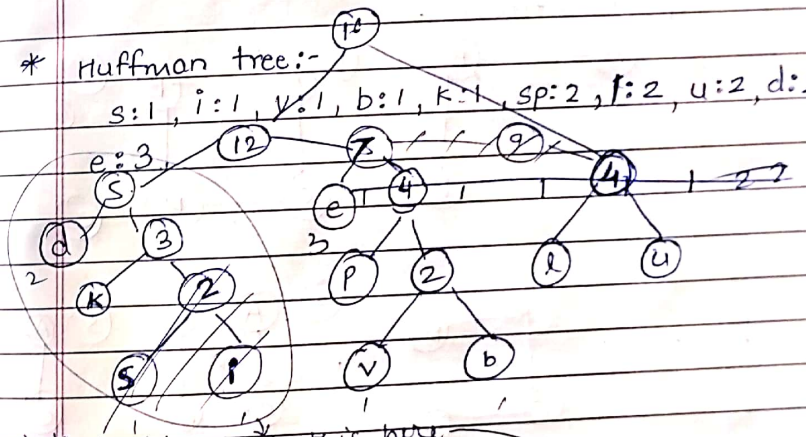


- Postfix to Expression Tree Algorithm,
  Eg. 2, 6, *, 3, 8, /, +,



* Huffman tree:-

s:1, i:1, v:1, b:1, k:1, sp:2, f:2, u:2, d:2,



Note: We wrote this here Properly.