Trees

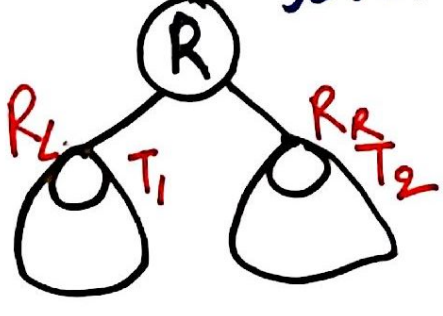
Strings
arrays
lists
stacks

* Till now several linear d.s we have seen.
* Tree is a non-linear datastructure!
* Majorly used to represent hierarchical relationships.
* In linked list each node can point to another node. (only one)
* In Tree one node may point to several other nodes hence it is more flexible and powerful data structure.

Binary Tree: Binary tree (T) defined as finite set of elements (nodes) s.t

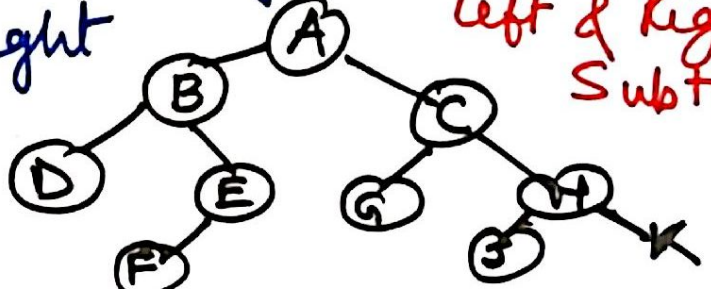
(a) T is empty (null tree or empty tree)

(b) T has a root node (R); and remaining tree nodes form an ordered pair of disjoint binary trees T_1 and T_2.



*) R_L & R_R are left and right successor of R.

*) Nodes with No successor are terminal nodes.



Left & Right Subtrees.

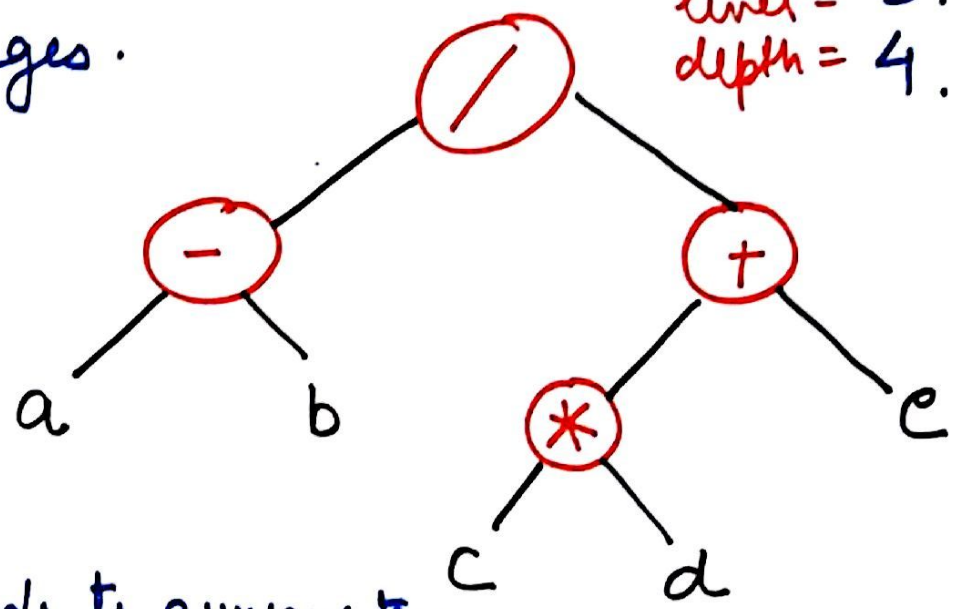
Scanned by CamScanner

Difference b/w Trees and Graphs :

- Tree is a minimally connected graph with only one path b/w any two vertices
- It is a special case of graph with no-loop/no-circuits/self loop.
- One root node and each child have only one parent.
- Tree traversal is a kind of special case of graph traversal. Pre | In | Post order traversal.
- Basically trees are Directed Acyclic Graphs
- Trees always has $n-1$ edges.

level = 3.
depth = 4.

$$E = (a-b)/((c*d)+e)$$



- The meaning of above expression is encoded in this tree
- Every algebraic expression corresponds to a unique tree.

Parent, Son, Siblings, Successor, Predecessor, Path, edge, leaf, level, depth

Scanned by CamScanner

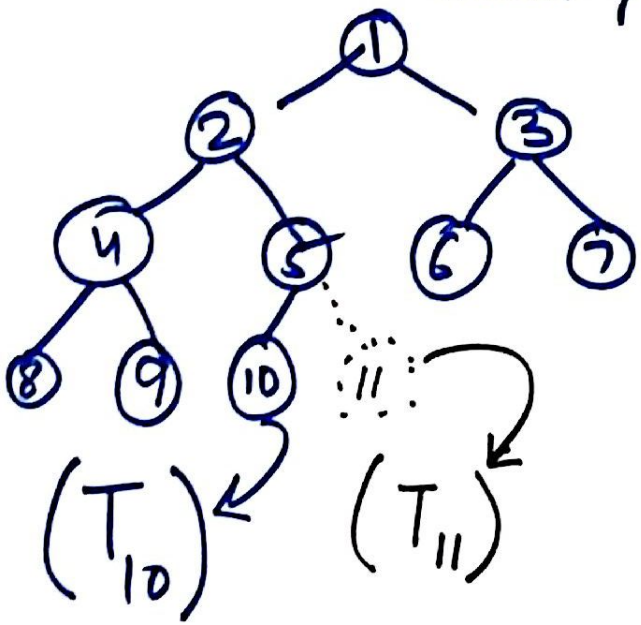
Complete Binary Tree:

*) Each node of (T) can have at most 2 children

*) Level (r) of T can have at most 2^r nodes.

*) Tree (T) is said to be complete if all its levels, except possibly the last, have maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible.

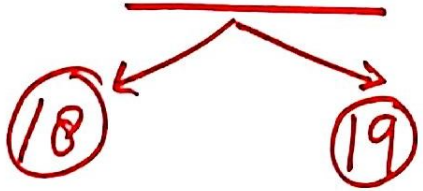
*) Hence structurally there is a unique complete binary tree T_n with exactly (n) nodes.



*) Left and right child of any node (K) are $(2*K)$ and $(2*K+1)$

*) Parent of (K) is node $\lfloor K/2 \rfloor$

Example: $K = 9$ → $\lfloor 4.5 \rfloor = 4$



Scanned by CamScanner

Binary Tree in memory:

[A] Linked representation: Very similar to linked list

* It uses 3 parallel arrays, INFO, LEFT and RIGHT.
* and there is a pointer variable as ROOT, pointing to root of (T).
* Each node (N) of (T) correspond to a location (K) s.t:
  - → INFO[K] - data at the node (N)
  - → LEFT[K] - location of left child of node (N).
  - → RIGHT[K] - location of right child of node (N).
* For empty subtree pointer contain NULL value.
* If ROOT = NULL ⇝ Tree is empty.

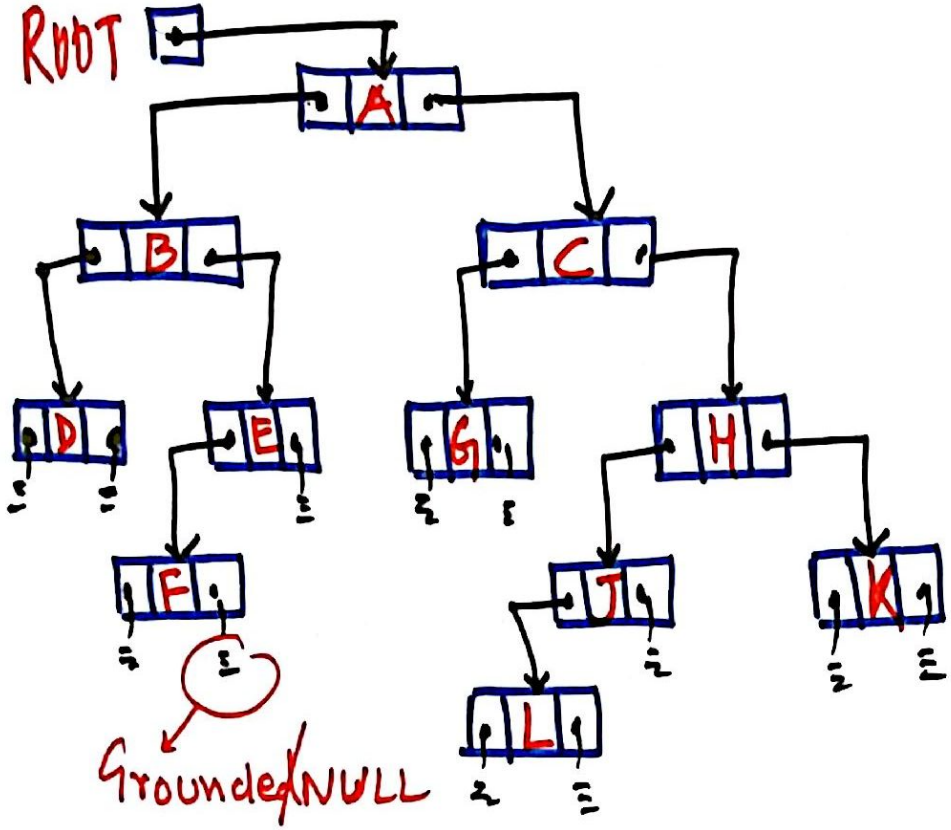
Garbage Collection

* Linked representation assumes insertion & deletion of nodes.

*) Hence free pool management must be done as a special list, pointing to all free nodes and accessed by pointer AVAIL.

*) AVAIL pointer to first empty node that subsequently points others.

Scanned by CamScanner



ROOT

5

AVAIL

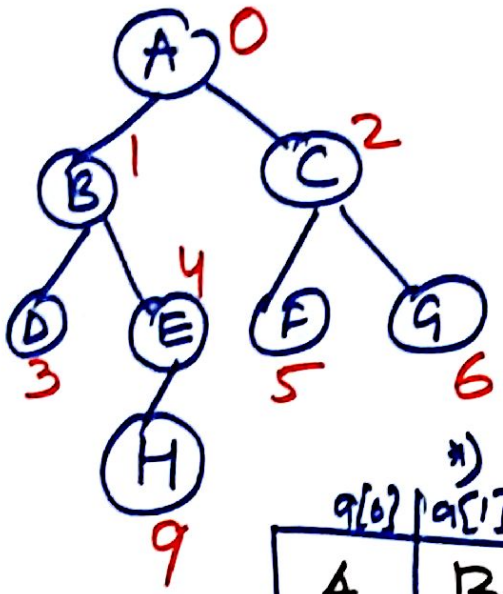
8

	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	J	7	0
18	D	0	0
19		20	
20		0	

AVAIL list is maintained as a one way list using the array LEFT.

Scanned by CamScanner

[B] Array based representation : Efficient when (T) is nearly complete



*) numbering left to right.

*) Root is 0th node.

*) So for any (K)

- Parent $\lfloor \frac{K}{2} \rfloor - 1$
- Left $2K+1$
- Right $2K+2$

Here an array (a).

*) [9] array must be defined like this.

a[0]	a[1]	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	B	C	D	E	F	G	\0	\0	H	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0

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

fn: Calling → Struct node *root;

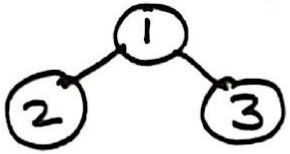
root = build tree(0);

```
struct node * buildtree(int n)
{
    struct node *temp = NULL;
    if (a[n] != '\0')
    {  temp = (struct node *) malloc(size of (struct node))
    temp->left = buildtree (2*n + 1);
    temp->data = a[n];
    temp->right = build tree (2*n+2);
    }
    return temp; }
```

Scanned by CamScanner

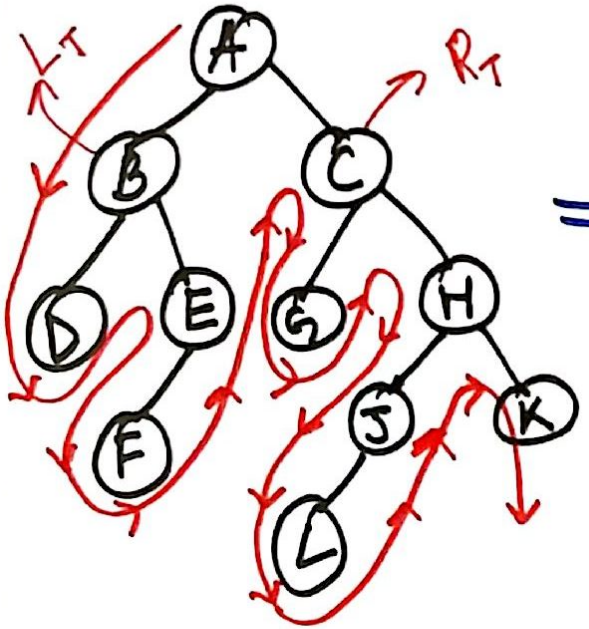
Tree traversals : 3 types of tree traversals are possible.

1. **Pre-order** →
   a) Visit root
   b) Traverse left sub-tree in pre-order
   c) Traverse right sub tree in pre-order
2. **In-order** → Left subtree, root, Right subtree
3. **Post-order** → Left subtree, Right subtree, root.



⟹

Pre -	1, 2, 3
In -	2, 1, 3
Post -	2, 3, 1



⟹

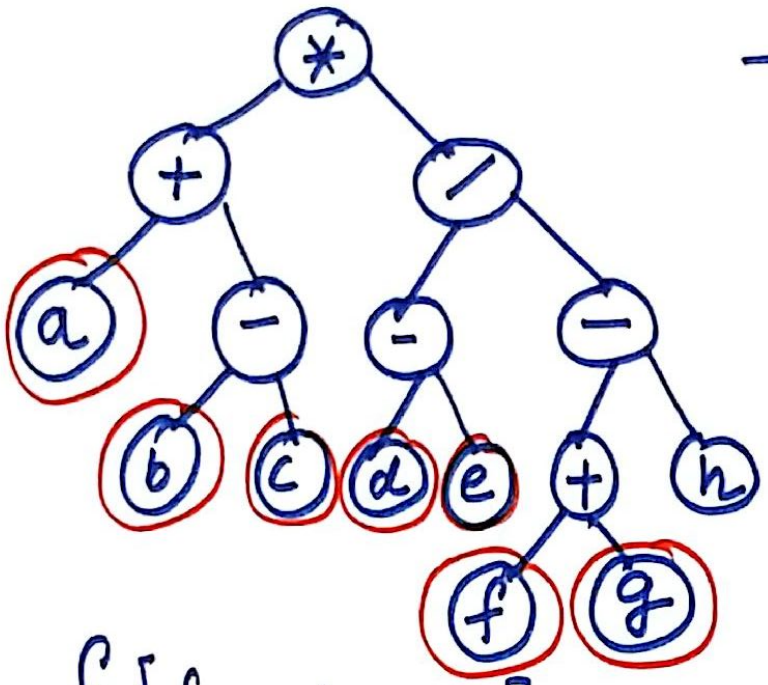
Pre : A, B, D, E, F, C, G, H, J, L, K.

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

Post : D, F, E, B, G, L, J, K, H, C, A.

Scanned by CamScanner

Expression evaluation: E : [a+(b-c)]*[(d-e)/(f+g-h)]



In :

Pre : *, +, a, -, b, c, /, -, d, e, -, +, f, g, h

Post : a, b, c, -, +, d, e, -, f, g, +, h, -, /, *.

fn [for inorder]

```
void inorder (struct node *root)
{
    if (root != NULL)
    {
        inorder(root -> left);
        printf("%c \t", root -> data);
        inorder (root -> right);
    }
}
```

(Recursive fn)

Change order to get others

Binary Search Tree: One can search in an average of $\log_2 n$

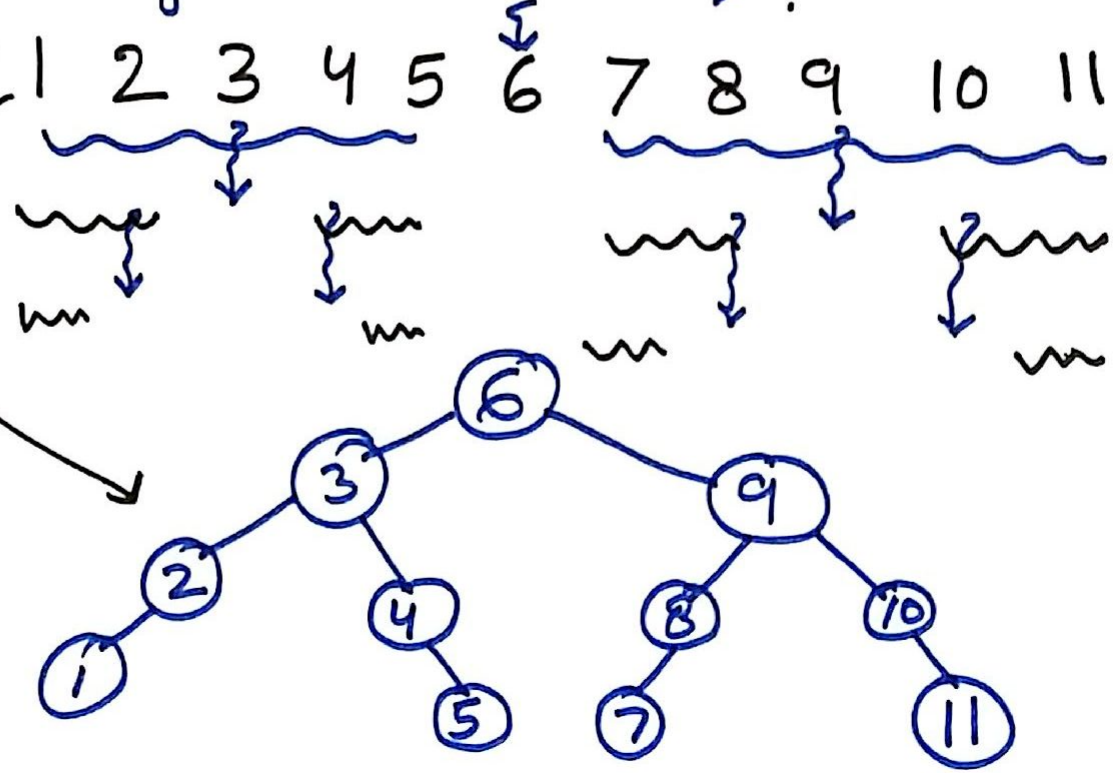
(T) Can be called as a BST, if each of its node N has following property:

↳ The value at (N) is greater than every value in the left subtree of (N) and is less than every value in the right subtree of (N).

(In order yields ~~~~~~).

↳ Just see this inorder: 1 2 3 4 5 6 7 8 9 10 11

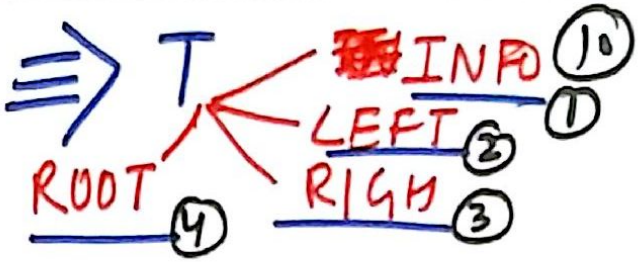
Searching using Binary Search gives us a tree structure



Inorder gives Sorted Sequence.

Scanned by CamScanner

BST Search and Insertion : Binary S.T $\Rightarrow$ T



*) One has to search an **ITEM** (5) in T;
and return its location **LOC** (6) and its parent location **PAR** (7)

3 Conditions

*) (a) LOC = NULL & PAR = NULL Tree(T) is empty.

(b) LOC $\neq$ NULL & PAR = NULL ITEM is root of (T).

(c) LOC = NULL & PAR $\neq$ NULL ITEM is **NOT** present in (T), But can be added to T as a Child of node(N) with location **PAR**

Scanned by CamScanner

FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

1. If ROOT = NULL (Tree is empty)
   LOC = NULL
   PAR = NULL and return.

2. If ITEM = INFO[ROOT] (ITEM is at root)
   LOC = ROOT
   PAR = NULL and return.

3. If ITEM < INFO[ROOT] then (pointer initialize)
   If ITEM < INFO[ROOT], then
   PTR = LEFT[ROOT]
   SAVE = ROOT
   Else
   PTR = RIGHT[ROOT]
   SAVE = ROOT

working using two pointers PTR & SAVE

4. While (PTR ≠ NULL)
5. If ITEM = INFO[PTR] (ITEM is found just return LOC, PAR.)
   LOC = PTR
   PAR = SAVE & Return

Scanned by CamScanner

6. If ITEM < INFO[PTR] then:

SAVE = PTR

PTR = LEFT[PTR]

ELSE

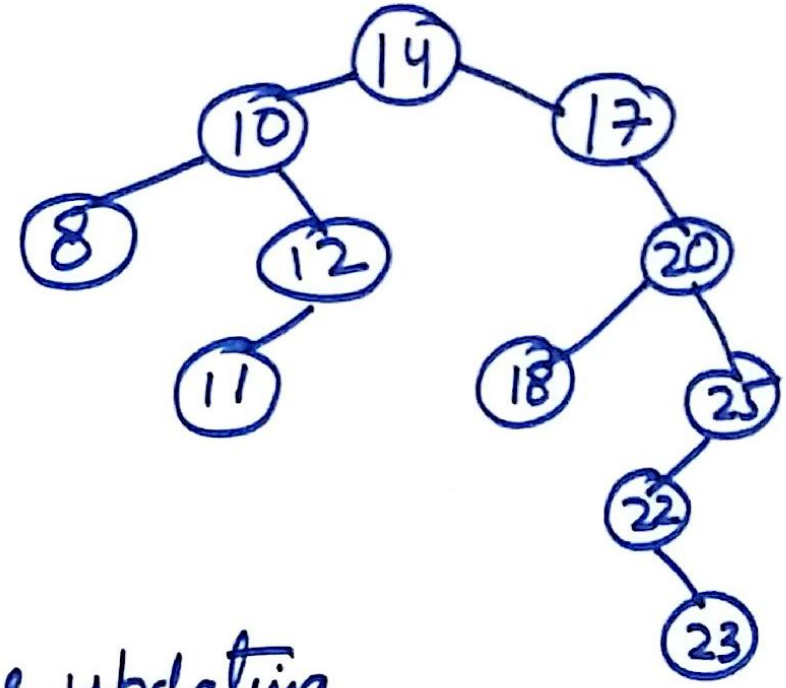
SAVE = PTR

PTR = RIGHT[PTR]

(End of while)

7. LOC = NULL

PAR = SAVE



Dry run algo to find

18 22 11 55 70

How SAVE & PTR pointers are updating.

Scanned by CamScanner

INSBST(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)

This algorithm finds the location LOC of ITEM in (T) or adds ITEM as a new node in T at location LOC.

1. FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
2. If LOC ≠ NULL, then Exit.
3. (a) If AVAIL = NULL
   Handle overflow exit.

   (b) New = AVAIL
   AVAIL = LEFT[AVAIL]
   INFO[NEW] = ITEM

   (c) LOC = NEW
   LEFT[NEW] = NULL
   RIGHT[NEW] = NULL

Copy ITEM into new node in AVAIL list.

Scanned by CamScanner

4. If PARA = NULL, then

ROOT = NEW

Else if ITEM < INFO[PAR] then

LEFT[PAR] = NEW

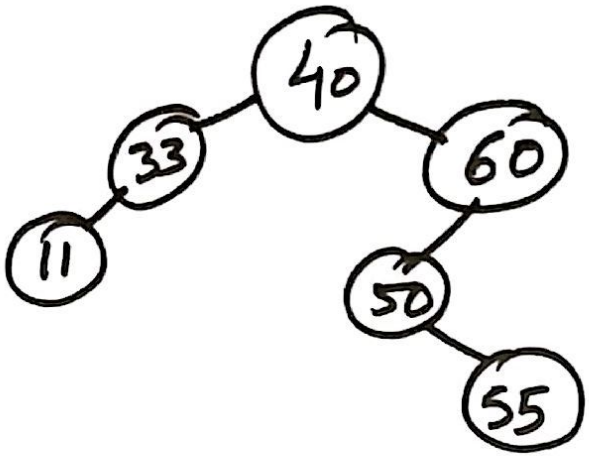
Else

RIGHT[PAR] = NEW

End of If

5. Exit.

Insert 40, 60, 50, 33, 55, 11



Scanned by CamScanner

Complexity (searching) For searching an element in (T) the number of comparisons are bounded by the tree depth.

*) Let we have n data items $A_1, A_2 — A_N$

*) There are $n!$ ways to put them in a Tree $\Rightarrow$ have $T_1, T_2 - T_{n!}$

*) It can be shown that average depth of $n!$ trees is

$$\frac{C * \log_2 n}{1.4 * \log_2 n}$$

Hence running time $\propto [\log_2 n]$

One application can be deleting all duplicates. $(A_1 — A_n)$

(A) $\rightarrow$ Basic algorithm $\rightarrow O(n^2)$

(B) $\rightarrow$ For $A_K \rightsquigarrow$ check till $A_{K-1} \rightarrow O(n^2)$

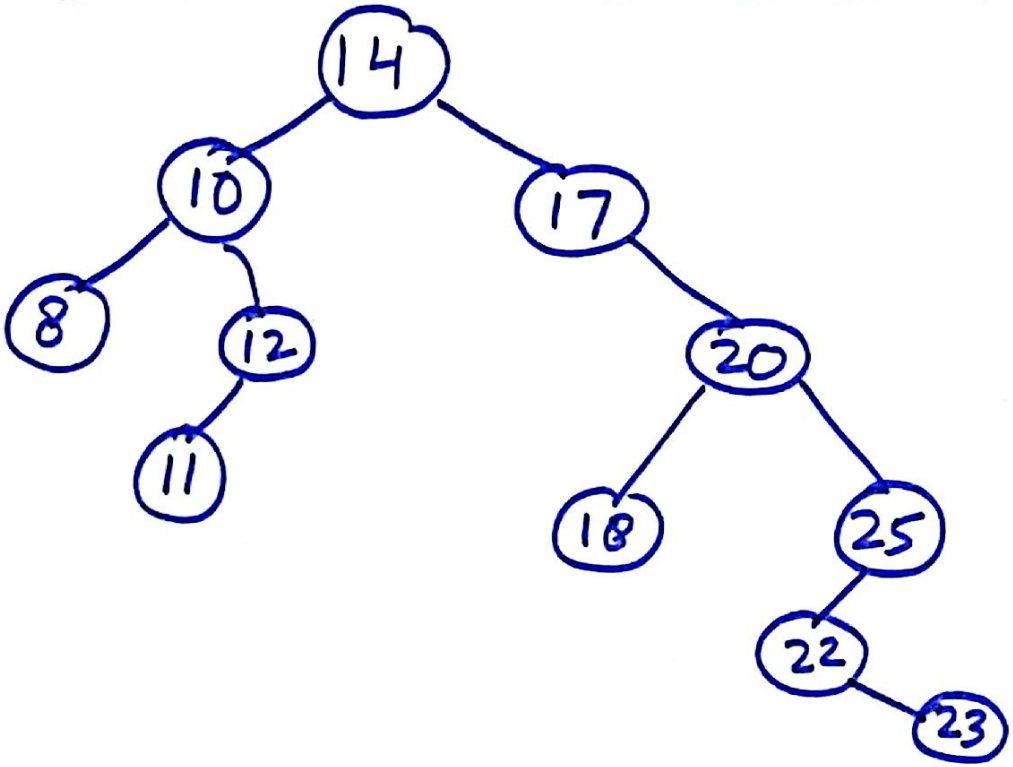
(C) $\rightarrow$ Build binary tree (T) using elements $A_1, A_2, — A_N$ during that delete A_K from the list if it already is present.

Scanned by CamScanner

14	10	17	12	10	11	20	12	18	25	20	8	22	11	23
0	1	1	2	2	3	2	3	3	3	3	2	4	4	5

$\Rightarrow \boxed{38}$

Algo (C)



Algo (B)

$\boxed{72}$

Scanned by CamScanner

Deletion from binary tree : How to delete an ITEM from tree T.

⊛ We use FIND function to first find the location of the node N (containing ITEM) and also its parent node P(N)

*) Deletion of $\boxed{\text{node } N}$ depends on number of Children of of node N. ⟹ [0, 1, 2] Three Cases.

Case 1 -: No children. Simply delete N, by replacing location of N in the parent node P(N) by NULL pointer.

Case 2 -: Exactly 1 Children. Delete N, by replacing location of N in P(N) by the location of the only child of N.
if left child ⟹ make left child or same as RIGHT.

Scanned by CamScanner

Case 3 : N has two children.

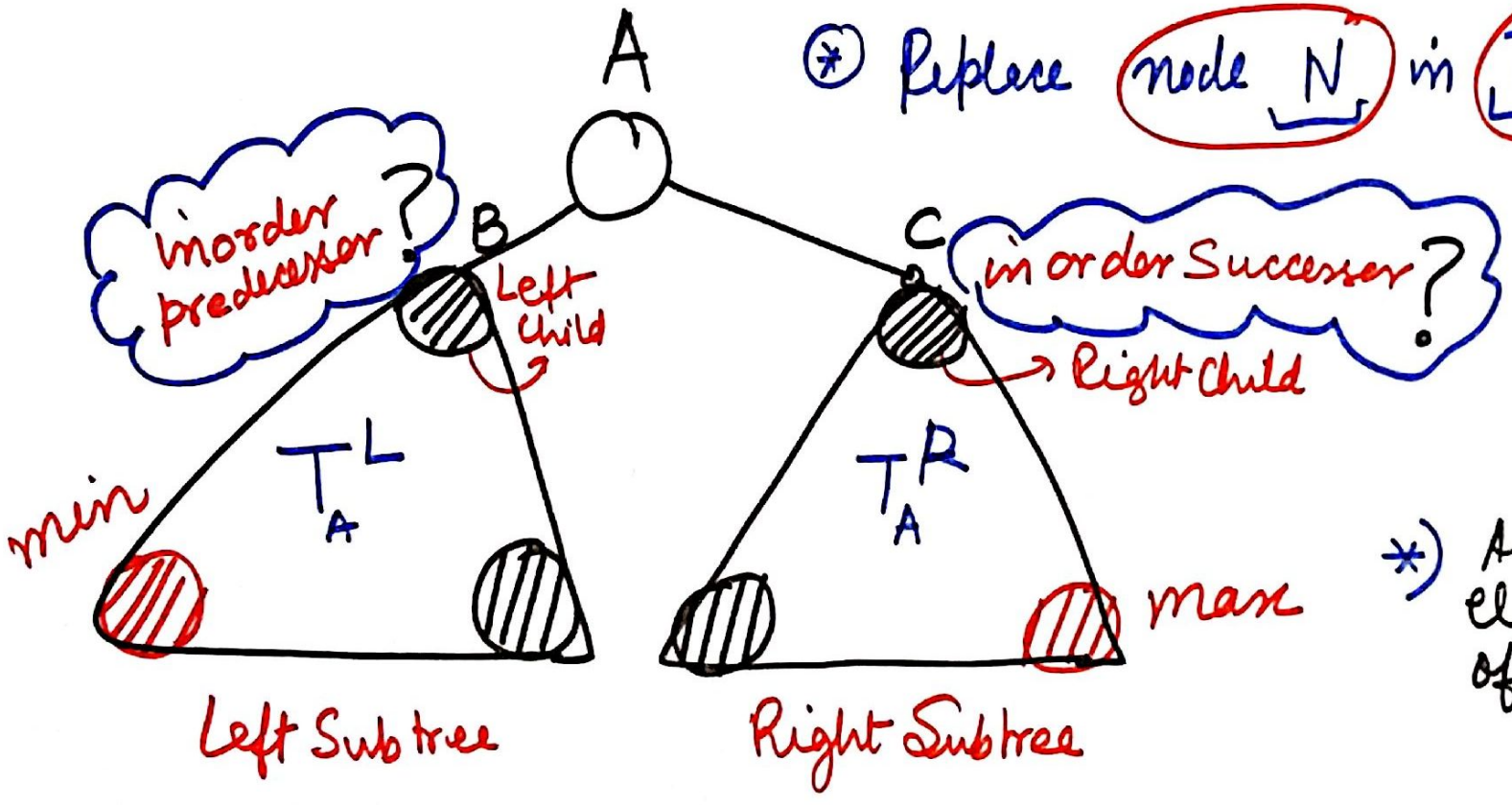
How to get max, min of any tree or Sub tree

→ Let S(N) denote inorder successor of (N).
(S(N) does not have left child)

→ Now N node can be deleted from T

⊛ first delete S(N) from T
(using Case 1 & Case 2)

⊛ Replace node N in T by node S(N)



*) $B < A < C$

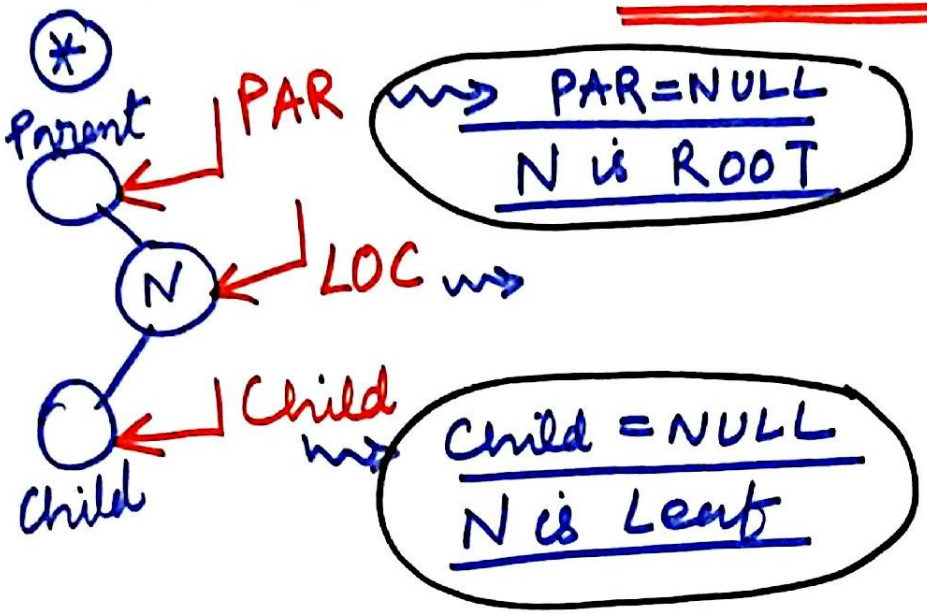
*) All elements of T_A^L < A < All elements of T_A^R

Scanned by CamScanner

CASE A (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

(*) Procedure to delete node N, at location LOC, when it does not have TWO children.

(*)



(1) Child initialization

If LEFT[LOC] = NULL & RIGHT[LOC] = NULL then
    CHILD = NULL.
Else if LEFT[LOC] ≠ NULL then
    CHILD = LEFT[LOC]
Else
    CHILD = RIGHT[LOC].
End if

(2) If PAR ≠ NULL then:
    If LOC = LEFT[PAR]
        LEFT[PAR] = CHILD
    Else
        RIGHT[PAR] = CHILD
Else
    ROOT = CHILD.
    RETURN.

Scanned by CamScanner

2 Children Case — CASEB(INFO, LEFT, RIGHT, ROOT, LOC, PAR)

* Delete node N at LOC with 2 children.
* SUC pointer gives the location of inorder successor of N
* PARSUC → pointer to parent of inorder successer.

1. Find SUC and PARSUC

(a) PTR = RIGHT[LOC] and SAVE = LOC.

(b) repeat while LEFT[PTR] ≠ NULL

SAVE = PTR & PTR = LEFT[PTR]

end of loop.

(c) SUC = PTR and PARSUC = SAVE

2. Delete inorder successor

CASEA(INFO, LEFT, RIGHT, ROOT, SUC, PARSUC)

Scanned by CamScanner

3 Replace node N by its inorder successor

(a)

```
IF PAR ≠ NULL then
    If LOC = LEFT[PAR] then
        LEFT[PAR] = SUC
    Else
        RIGHT[PAR] = SUC
    end of If
Else
    ROOT = SUC
```

(b)

```
LEFT[SUC] = LEFT[LOC]
RIGHT[SUC] = RIGHT[LOC]

RETURN
```

Scanned by CamScanner

DEL(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)

Binary Search Tree (T) — To be deleted.

① FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

ITEM is at LOC with parent PAR.

② ITEM is present or NOT.

If LOC = NULL, ITEM is Absent and Return.

③ Delete node Containing ITEM.

If RIGHT[LOC] ≠ NULL and LEFT[LOC] ≠ NULL then (Both Children are present)

CASE B(INFO, LEFT, RIGHT, ROOT, LOC, PAR)

Else

CASE A(INFO, LEFT, RIGHT, ROOT, LOC, PAR)

end of IF.

④ Return deleted node to AVAIL LIST

LEFT[LOC] = AVAIL and AVAIL = LOC

⑤ RETURN.

Scanned by CamScanner