~~Generating Linked Binary Trees~~

Binary Search Tree

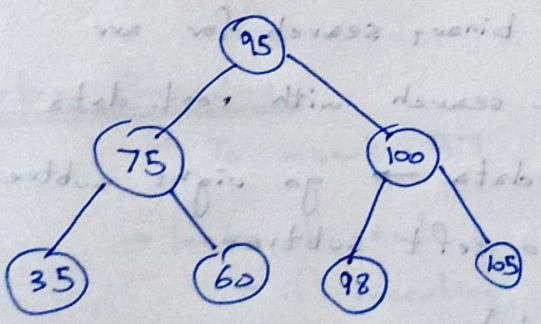
→ A Binary Search Tree t is a binary tree, which is either empty, or each node in the tree contains an identifier and

i) all identifiers in the left subtree of t are less than the identifiers ~~in the right~~ in the root node of ~~tree~~ t

ii) all identifier in the right subtree of t are more than the identifier in the root

iii) The left-subtree and the right-subtrees are also binary seach tree (so the ~~de~~ tree is recursive.)

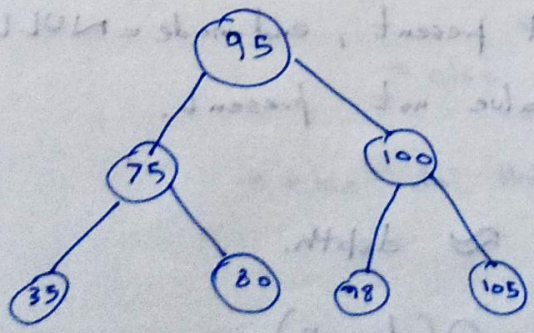
~~Eg~~:



→ This is not Binary Search Tree

→ 60 < 75 but right subtree should be more than node.

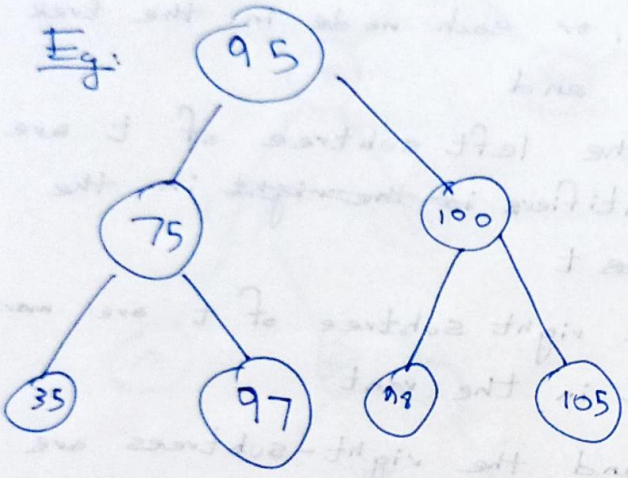
~~Eg~~:



→ This is a Binary Seaurch Tree.

→ if we want to include equal value, we must modify definition to account for equality.

Eg:



→ This is not a binary search tree as 97 > 95 but all value in left subtree should be less than node.

→ This tree is used for searching elements.

Binary Search Tree Searching

→ This is similar to binary search for arr

→ check value to be search with root data

if value > data → go right subtree

else → go left subtree

(do it recursively)

→ if value not present, end node = NULL,

~~search~~ value not present.

→ no. of comparision ≈ depth.

→ For a balanced tree, $O(\log_2 n)$

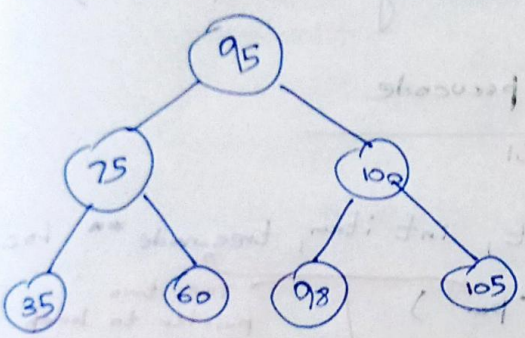
→ For skewed tree, $O(n)$

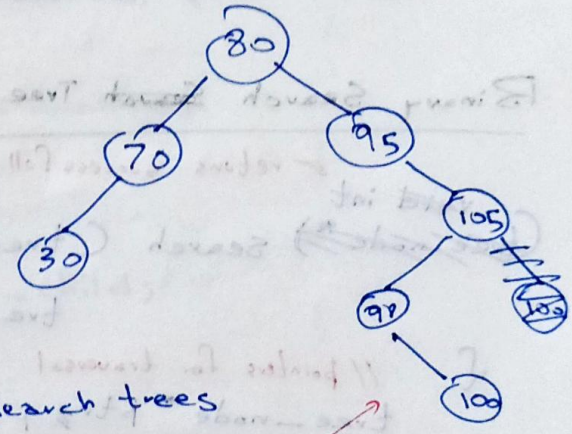
→ for a Binary Search tree

→ Inorder Traversal outputs Ascending list

→ for a set of number / identifiers, we can have multiple Binary Search Trees

Eg: for identifiers = {35, 75, ~~97~~ 80, 95, ~~100~~ 98, 100, 105}





are both binary search trees

Inserting Procedure

→ To insert 97 in the tree

→ look at node

→ if inserting value > node.

→ if (node → r_tree == NULL)

set r_tree = value

→ else

recurse with right-tree.

→ else do the ~~same~~ similar thing with left-tree

→ we also need to keep track of the previous node (to assign new element, incase curr = NULL)

→ so we use two pointers to node
  → current node pointer
  → previous node pointer.

(all this is ~~assum~~ done assuming no duplicates)

Binary Search ~~Search~~ Tree searching pseucode

← retuns successfull or unsuccessful

```
(~~void~~ int) search (tree_node *t, int item, tree_node **loc,
                       tree_node **par)
{   // pointers for traversal
    tree_node *ptr, *save;

    if (t == NULL){
        *loc = NULL;
        *par = NULL;
    }
    if (t->data == item) {
        *loc = t;
        *par = NULL;
    }
    // initialising ptr and save
    if (item < t->data) {
        ptr = t->l_child;
        save = t;
    } else {
        ptr = t->r_child;
        save = t;
    }
}
```

The two pointer to keep track

double pointer so as to treat it as return values

```
while (ptr != NULL ) {
    if (ptr -> data == item) {
            *loc = ptr;
            *par = save;
            return 0;
    }
    if ( item < ptr -> data) {
            save = ptr;
            ptr = ptr -> l_child;
    }
    else {
            save = ptr;
            ptr = ptr -> r_child;
    }
} //end of while
*loc = NULL;
*par = save ;
return -1; //element not found
} //end of function
```

This is done so as to differentiate between Empty Tree and failed search

→ now *loc and *par can be used to insert new element.

Cases

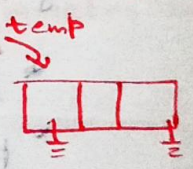
1) loc = NULL par = NULL → Tree empty
2) loc = NULL par != NULL → element not there.
3) loc != NULL *par = NULL → root node.
4) loc != NULL par != NULL → element found. (no need to insert, may need to edit node)

Insert pseudo code

```
int insert (tree_node *t, int item)
{
    node *loc, *par, *temp;
    int res = search (t, item, &loc, &par);
```

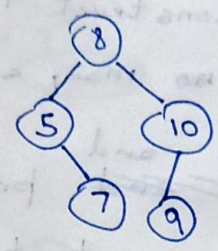
call by pointer to save loc & par ~~after~~ inside fn.

```
    if // res can be checked, ignoring

    if ( loc != NULL) {
        printf (" Item already exist ");
        return 0;
    }

    temp= new ( ) ; //assume creates a node for tree
```




```
    temp -> data= item;
    temp -> l_child = NULL;
    temp -> r_child = NULL;

    if (par ==NULL) //tree empty
        t= temp;
    else if (item < par -> data)
        par -> l_child = temp;
    else
        par -> r_child = temp;
} //end of fn.
```

Eg: inserting {8, 5, 10, 7, 9} in order.

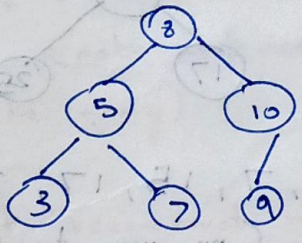


~~Eg:~~ Eg: insert 3 in previous tree.

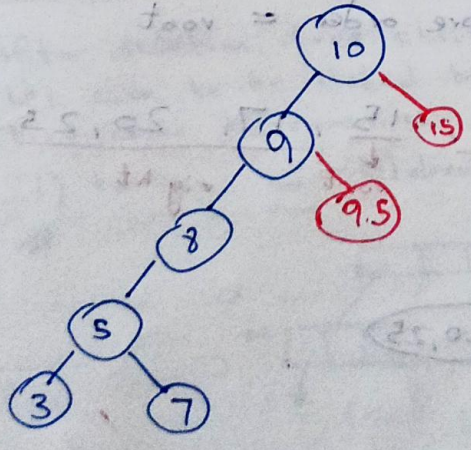
→ loc = NULL

→ par = 5 (not actually, but for simplification)

∴ tree:



Eg: changing order of previous inputs to {10, 9, 8, 5, 3, 7}



→ hence changing order changes the tree.

(first element in order will become root in Binary Search Tree)

Eg: insertion 9.5 to prev tree.

loc = NULL

par = 9

Eg: insertion (15) to prev tree:

loc = NULL

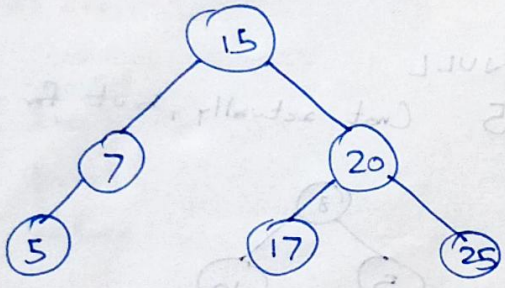
par = 10

Reconstruction of Tree

→ if given a in-order traversal of a tree, it is impossible to reconstruct the exact binary tree, as there can be many solution.

→ if given in-order and pre-order traversal list, we can get the exact tree,

Eg:



in-order = {5, 7, 15, 17, 20, 25}

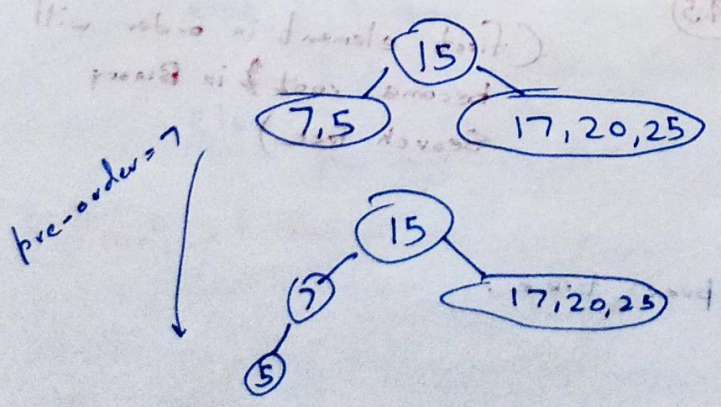
(no way to reconstruct tree from in-order)

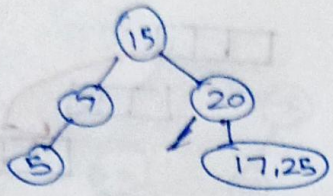
pre-order = {15, 7, 5, 20, 17, 25}

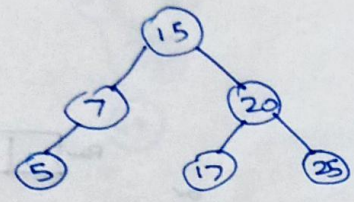
→ first element of pre order = root

∴ in-order = 5, 7, 15, 17, 20, 25

left | root | right







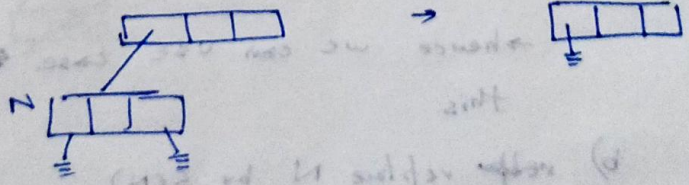
→ ~~no way to reconstruct from pre-order alone~~

Q) ?? → given only pre-order, we can reconstruct binary Search Tree, (not applicable to normal tree)
(This can be wrong)

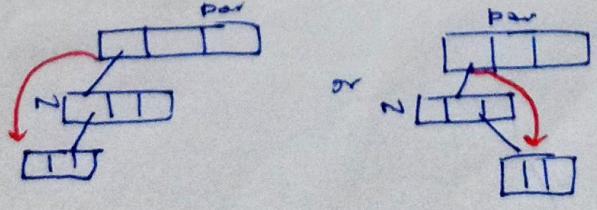
Sir says: we can't reconstruct with only pre-order
(will give counter-example)

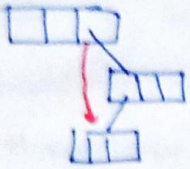
Deletion in BST

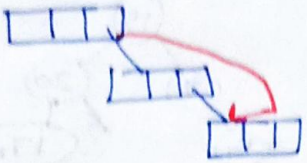
→ After deletion, tree should be BST
→ Let node to be deleted be N
→ Three cases to consider
i) N has no children



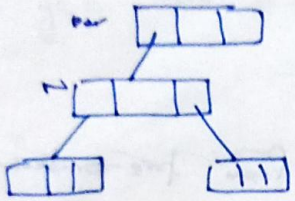
ii) N has exactly one child.



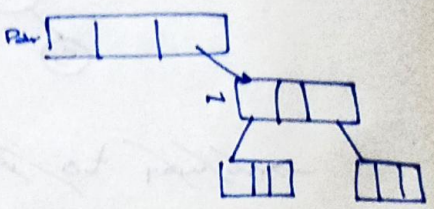




iii) n has two children

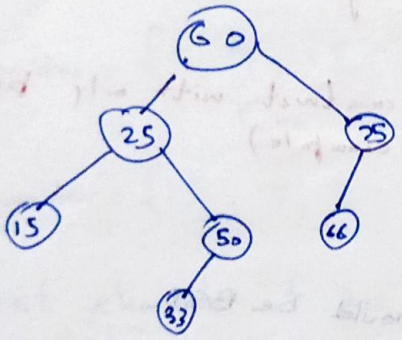


or



→ find in-order succor of n (let it be S(N))

Eg:



→ in order succesor of 60 is 66

→ in order:

15, 25, 33, 50, 50, 60, 66, 75

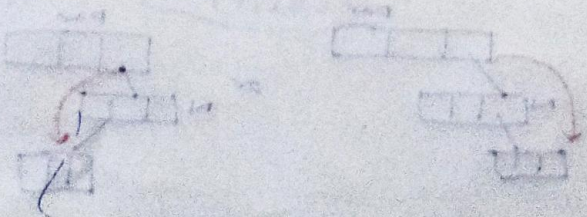
→ to delete N

a) → first delete S(N) → cannot have left child

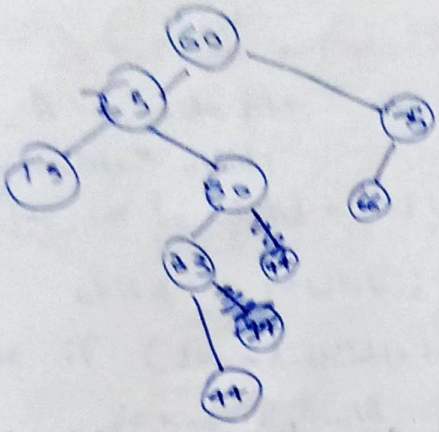
→ can have at most 1 r-child

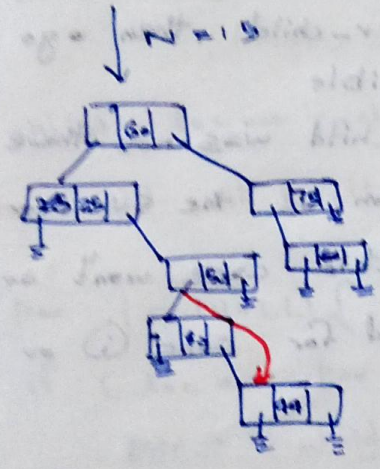
→ hence we can use case (i) or case (ii) to do this

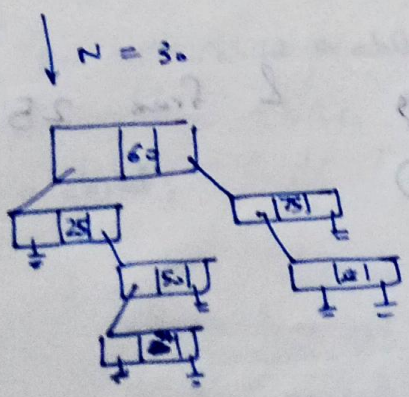
b) replace N by S(N)



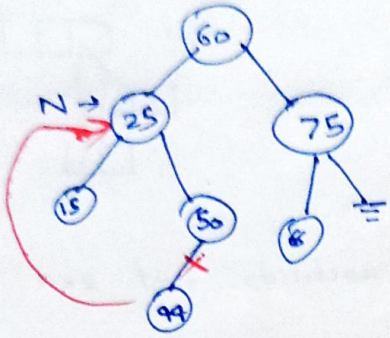
Eg







Eg:

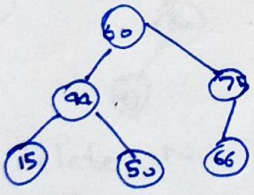


→ to ~~go t~~ find in-order successor is BST

→ go one r_child, then go l_child as far as possible

→ if r_child was not there, it can cause problem, as the successor will be parent, but this case won't arrive if we had checked for case (i) or (ii) before.

end result



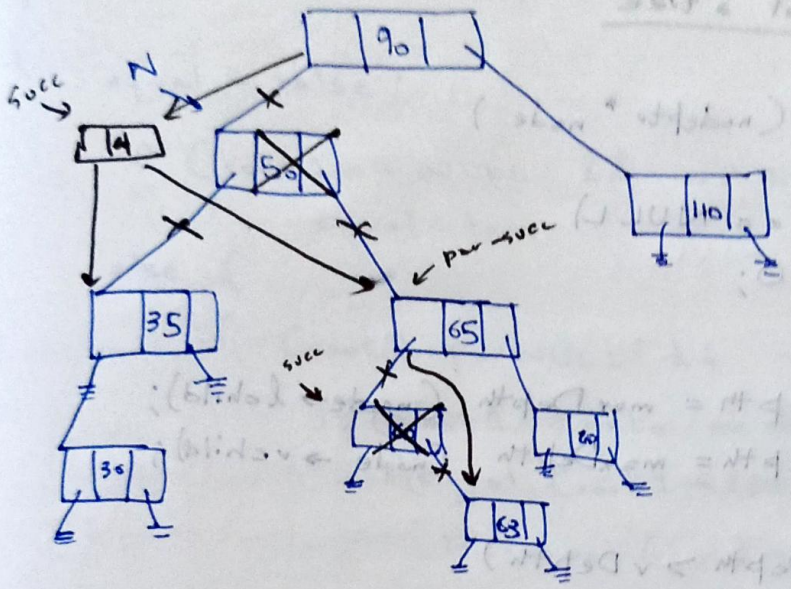
& free 25 node

pseudo code → has no or one child.

```
case_A (nodeptr * root, loc, par, item)
{
    nodeptr* child;
    if (loc -> l_child = NULL && loc -> r_child == NULL)
    {
        child = NULL;
    }
    else if (loc->l_child != NULL) {
        // has l_child
        child = loc -> l_child;
    }
    else
        child = loc -> r_child;
    // the child will be attached to par.
    if (par != NULL){
        if (loc == par->lchild)
            par -> l_child = child;
        else
            par -> r_child = child;
    }
    else
        root = child;
}
```

→ has two child

```
case_B ( node ptr *root, *loc, *par, *item ) {

    node ptr * suc , *par suc , *ptr, *save;

    ptr = loc → r_child;

    while (ptr → l_child != NULL) {
            save
            ~~suc~~ = ptr;
            ptr = ptr → l_child;
    }

      suc = ptr;
      par_suc = ~~suc~~ save;

      case_A (root, loc, suc, par_suc);

       if (par != NULL) {
          if (loc = par → l_child)
                par → l_child = succ;
          else
                par → r_child = suc;
       }
       else
            root = suc

      suc → l_child = loc → l_child;
      suc → r_child = loc → r_child;

}
```

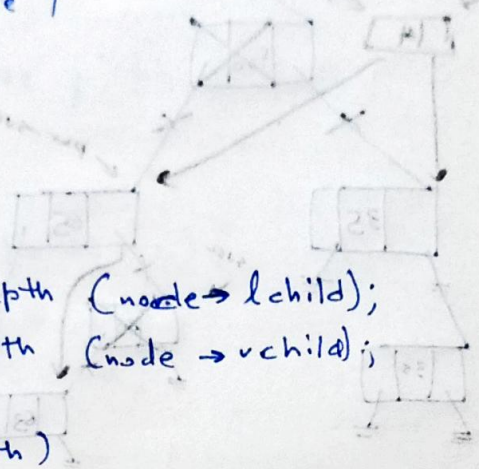


→ putting it all together.

```
void del (node ptr * root, item)
{
    search (root, item, loc, par)
    if (loc == NULL){
            printf ("item not there");
            return;
    }
    ~~else~~ ~~do~~
    if ( loc->lchild != NULL & loc->rchild != NULL )
        case_B (root, loc, par, item);
    else
        case_A (root, loc, par, item);

    free(loc);
}
```

Find the height of a tree

```
int maxDepth (nodeptr * node)
{
    if (node == NULL)
        return 0;
    else
    {
        int lDepth = maxDepth (node->lchild);
        int rDepth = maxDepth (node->rchild);

        if (lDepth > rDepth)
            return (lDepth + 1);
        else
            return (rDepth + 1);
    }
}
```



Given two trees given to you check if they are same.

Approach 1: Check any two of their traversal (too long)

Approach 2: ~~Depth First Search~~. One traversal check if equal, if not return false. (good one)

```
equal (nodeptr* root1, nodeptr* root2)
{
    equal = false.
    if (root1 == NULL && root2 == NULL)
        equal = true
    else {
        if (root1 != NULL && root2 != NULL)
            if (root1 -> data == root2 -> data) {
                if(equal (root1 -> lchild, root2 -> lchild )){
                    if (equal (root1 -> rchild, root2 -> rchild))
                        *equal = true.
                }
            }
    return ~~true~~, equal;
}
```

Copy tree

~~My try~~

```
nodeptr* copy (nodeptr* ~~root~~ node.) ~~nodeptr* par = NULL~~)
{
    nodeptr* copy;
    copy = ~~malloc~~ (nodeptr*) malloc (sizeof (nodeptr));
    copy -> data = node -> data.
    ~~if (par != NULL)~~ if (node -> lchild != NULL) {
        ~~par = par -> l~~
        copy -> lchild = copy (node -> lchild); }
if (node -> rchild != NULL){
        copy -> rchild = copy (node -> rchild); }

    return copy;
}

// not handled ~~return~~ return NULL
```

Correct code

```
struct node * cloneTree (struct node * root)
{
    if (root == NULL)
        return NULL;

    struct node * newNode = (struct node *) malloc (sizeof (node));
    newNode -> data = root -> data;
    newNode -> left = cloneTree (root -> left);
    newNode -> right = cloneTree (root -> right);

    return newNode;
}
```

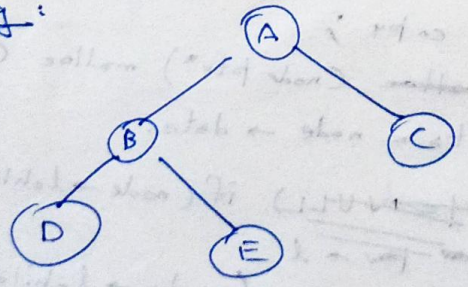
Traversal, Non-Recursive Version

I) Pre Order (DLR)

→ we used stack for recursion.

→ so we use stack

Eg:



Post order → D E B C A

Pre → A B D E C

Algo

→ initialize stack. , initial push NULL, set a ptr to root

→ then repeat following (until ptr != NULL)

{

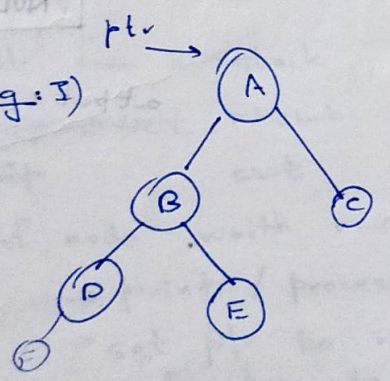
a) → proceed down the leftmost path by ptr, pushing each right child R(N), (if any) onto the stack.

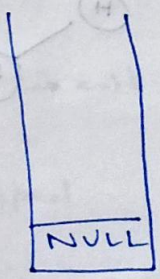
→ The traversing stops after a node N with no left child is encountered.

b) pop and assign ptr to top element, if ptr ≠ NULL then return to step (a) otherwise exit.

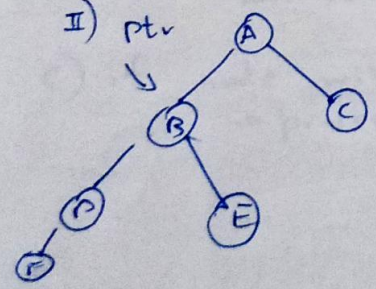
}

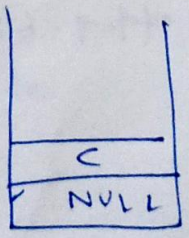
Eg: I)



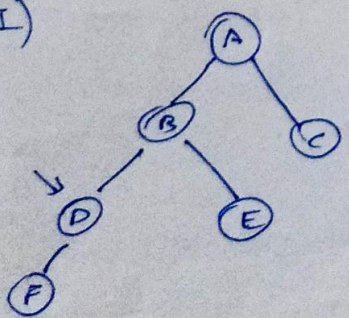


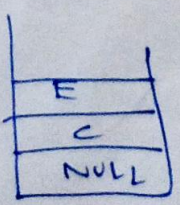
II)



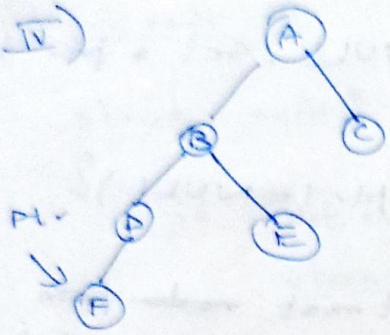


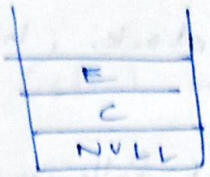
III)





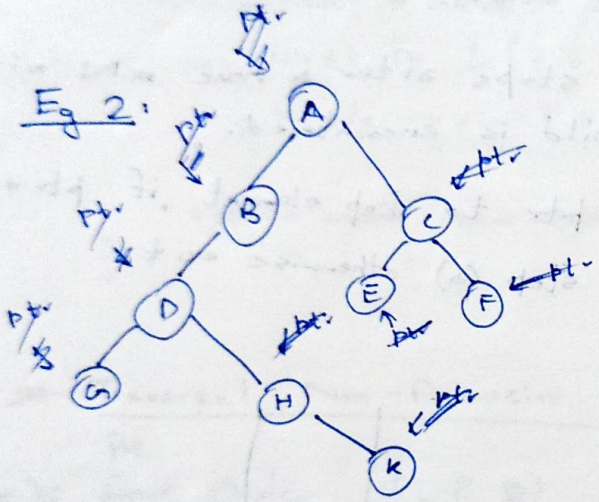
IV)

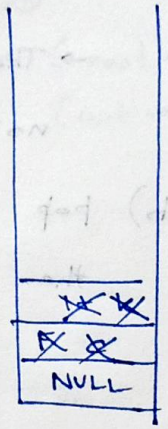




output : F

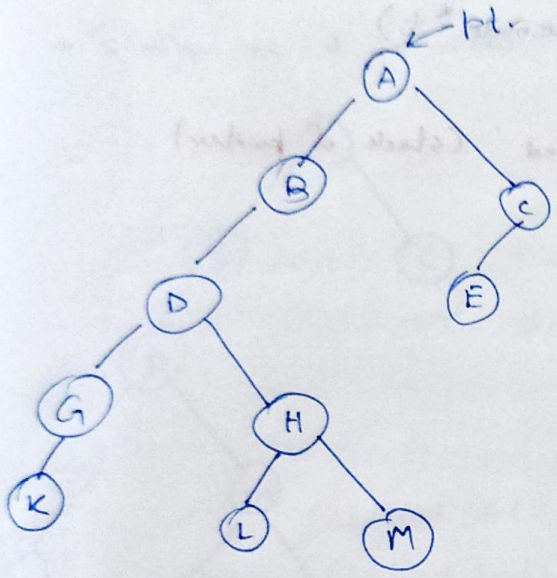
Eg 2:





output : A B D G H K C E F

II) In-order (L D R)



Inorder:
K G D L H M B A E C

Steps: 0) initialize stack with first element NULL

i) Proceed down left most path, pushing each ~~element~~ node encountered and stopping ~~when~~ ~~ptr~~ at a node which doesn't have l-child.

ii) pop a node from a stack

a) ~~pop NULL~~ NULL popped → exit

~~b) pop~~ ~~exit~~

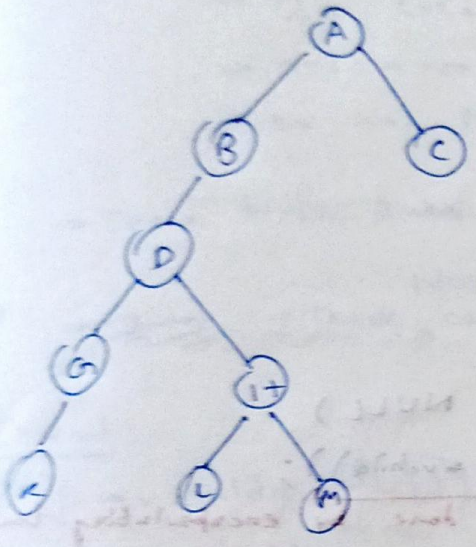
b) if node with r-child popped
→ print / process node.
→ set ptr to right child of that node. and return to step 4

c) if node with no r-child popped
→ process / print node.

```
void in_order_iterative (treenode *t)
{
    // assume stack already declared (stack of pointers)
    treenode *temp = t;
    while (1)
    {
        while (temp != NULL) {
            push (temp);
            temp = temp->l_child;
        }

        temp = pop();

        if (temp==NULL)
            break; //traversal stops

        printf (temp -> data);

        temp = temp -> r_child;
    }
```

II) Post order (LRD)

→ Similar as In-order.



K G L M H D B C A

→ as we have to keep present element in stack we have to distinguish r-childs

→ we do it by -node. (or anything else)

Steps

1) ~~go~~ proceed to left most treed, ~~adding~~ push all node and their r-childs (after node) with a special indicator.

2) Pop

i) if NULL end

ii) if pop r-child.
→ set ptr to r-child
→ do step 1 again.

iii) if ~~not~~ pop normal node
→ process/print node.

```
postorder (treenode *t)
{
  treenode* temp = t
  while (1)
  {
    while (temp != NULL)
    {
      push (temp);
      if (temp->rchild != NULL)
      push (-(temp->rchild));
      temp = temp->lchild;
    }
    temp = pop();
    if (temp == NULL)
      break;
    if (temp->flag == 0)
      print (temp->data);
    else
      temp = -temp;
  }
}
```

can be done by encapsulating temp with a bool in a structure.
or making a flag variable. inside node

not '-' flag.

if +ve already printed
if -ve, -(-temp) = temp

end of inner while.
end of function.

→ '-' doesn't effect data of node, it's just a flag

```
while (temp > 0)
{
  print (data)
  temp = pop()
}
temp = -temp;
}
```

Threaded Binary Trees

→ In a n node BST

→ 2n total links

→ n-1 → non-NULL links

→ So n+1 NULL links

→ These NULL links are waste of space.

→ ~~A.J.~~ → These NULLs can be made "threads"

Thread

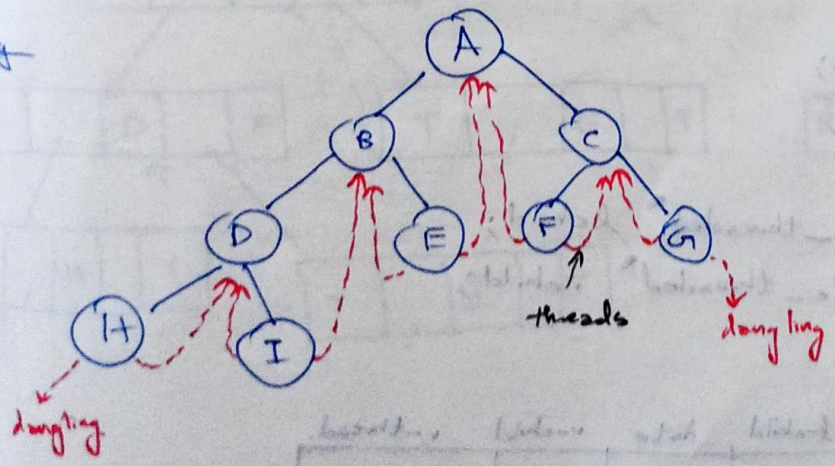
→ if (p→rchild = NULL)

→ we replace p→rchild by a pointer to a node which comes after p if we traverse the tree in in-order traversal.

→ if (p→lchild = NULL)

→ we replace p→lchild with a pointer to a node which come ~~after~~ before p in in-order traversal.

Eg



we have to differentiate between actual pointers and thread.

→ we do this by two bools
  → lthread
  → rthread.

→ if (p→lthread == True)
  p→lchild points to in-order predecessor.

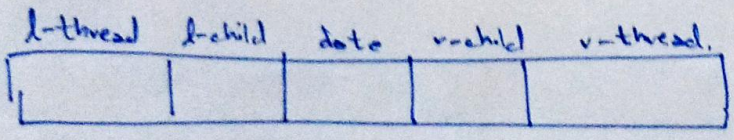
→ if (p→rthread == True)
  p→rchild points to in-order successor.

→ if (p→lthread == False)
  p→lchild points to valid node

→ if (p→rthread == False)
  p→rchild points to valid node.

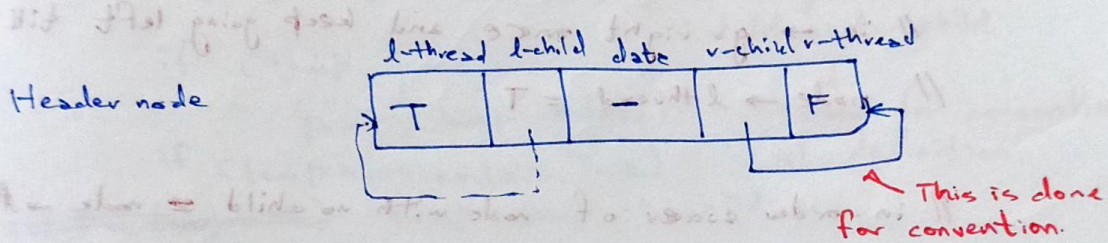
Structure definition

```
typedef struct treenode_threaded
{
    bool lthread;
    bool rthread;
    int data;
    struct treenode_threaded* lchild;
    struct treenode_threaded* rchild;
}
```

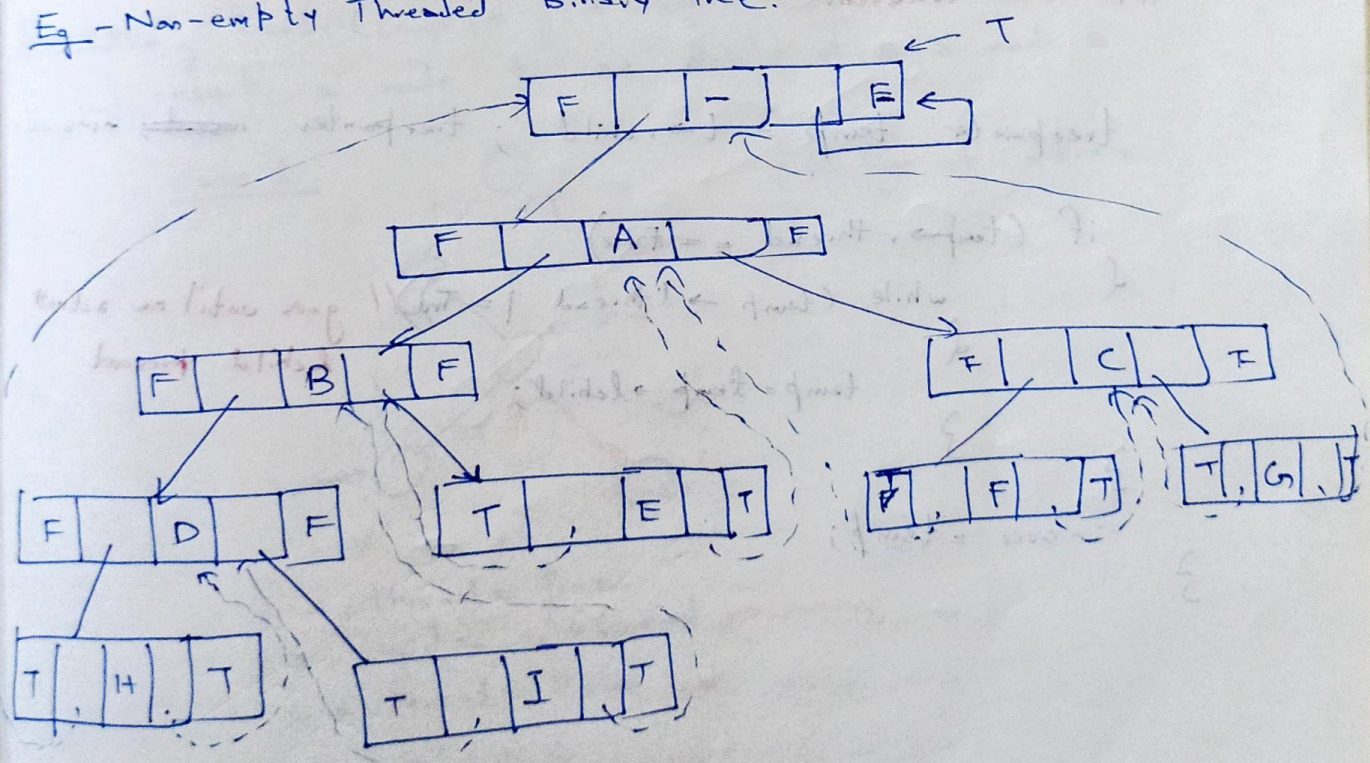


→ We observe that there exist two dangling pointer, to avoid this, we make a header node. , with root being l-child of header node, all the dangling pointer points to header node.

Eg: Empty threaded binary tree



Eg - Non-empty Threaded Binary Tree.



Inorder: H D I B E A F C G

→ Why Threading ?

→ finding in-order is very easy

~~treechild~~ ~~in~~ ~~order~~ insucc (treepointer t) ← the node for which you want to find in-order successor.

{

// in order successor of node with two child

// is → go right once and keep going left till ~~no child~~

// node → lthread = T

// in-order succer of node with no child = node → rchild.

// in-order successor of node with = lchild = node → rchild

treepointer temp = t → rchild ; treepointer ~~insert;~~ insucc;

if (temp → rthread == False)

{ while (temp → lthread != True) // goes until no actual lchild present

{ temp = temp → lchild;

}

}

insucc = temp;

}

In-order Traversal

header

```
In_order_Traversal (treeptr t)
{
  tree ptr temp = t, is_succ (t)
  repeat {
    temp = insucc (temp);
    if (temp != t)
      print (temp -> data)
    if (temp -> rthread == True)
      temp = temp -> rchild;
    else
      temp = in_succ (temp);
  } until (temp != gt -> header)
```

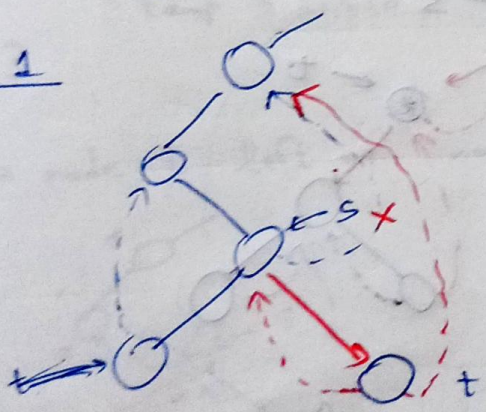
→ rchild of header is itself, then we go left till no actual lchild and rchild

↓ reason for convention of definition.

Insertion

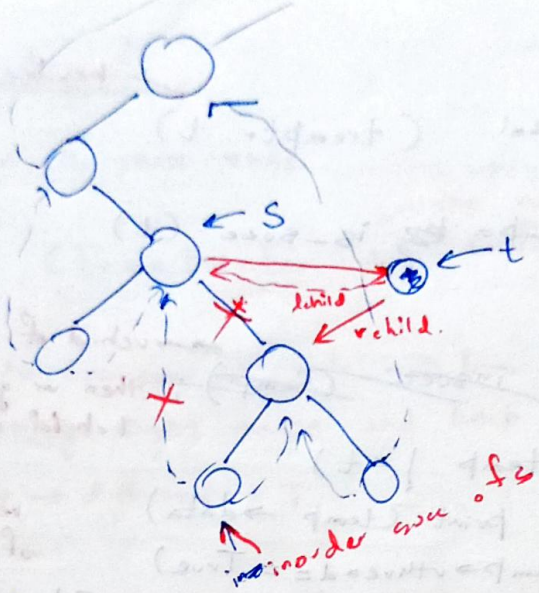
I) insert a node t as rchild of a node s

Case 1 (Portion of a tree

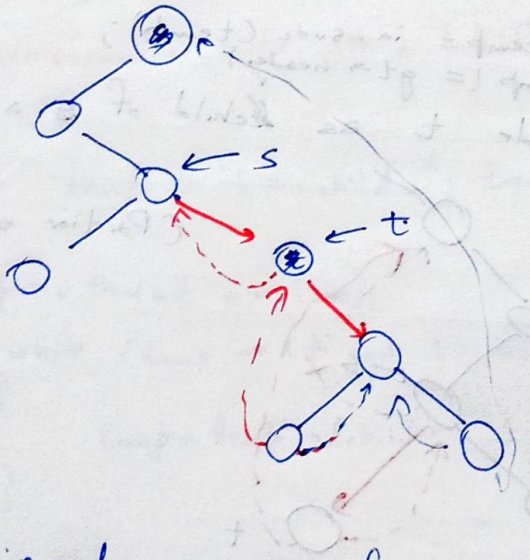


t → rthread = True.
t → rchild = s → rchild → common.
s → rthread = False
s → rchild = t
t → lthread = True
t → lchild = s

Case 2 :



end result



go to inorder succesor of s ⇒ k

```
t → rchild = s → rchild → common
s → rchild = t;
t → lthread = true;
t → lthread = s;
k → lthread = true;
k → lchild = t;
```

Algorithm

(make insert left fn. with diagram and for Homework till tomorrow)

insertright (tree ptr t, tree ptr s)

// insert t as r child of s

{

tree ptr = temp;

t → r child = s → r child;

t → r thread = s → r thread;

t → l child = s

t → l thread = True;

s → r thread = False;

s → r child = t

if (t → r thread == False)

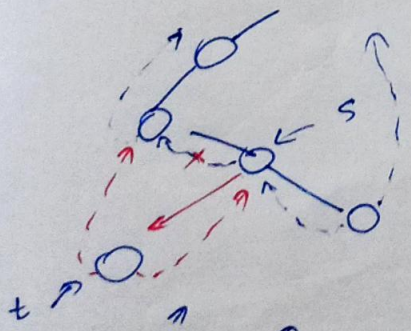
{ temp = insuc (t)

temp → l child = t // temp → l thread already true.

}

II) insertint a node t left of nodes.

Case I



t → l thread = s → l thread.

s → l child = t

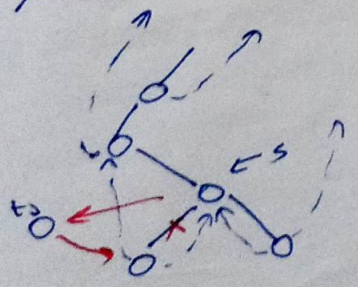
t → l child = s → l child.

s → l child = t

t → r child = s.

t → r thread = True.

Case II

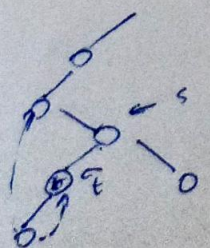


go to insuc(s)

go to inorder predecessor (s) = k

k → r child = t.

t → r child = s



→ go to inorder predecessor of s (i.e s → l child) = k

→ k → r child = t.

→ t → r child = s

Algorithm

```
// This function inserts node t as in left of node s
void insert_left ( treeptr s, treeptr t)
{
    treeptr inpre = s->lchild; //inorder predecessor of s
    t->lchild = s->lchild; //transfering s->lchild to t->lchild.
    t->lthread = s->lthread;
    t->rchild = s;          //threading t to s
    t->rthread = True;
    s->lthread = False;     // connecting s to t
    s->lchild = t;

    inpre->rchild = t;  // repairing thread of inorder in order
                        // predecessor of s
}
```