Some properties of binary trees:

Property 1: The maximum number of nodes on any level i is 2^i where $i \geq 0$.

Property 2: The maximum number of nodes possible in a binary tree of height h is $2^h - 1$

Property 3: The minimum number of nodes possible in a binary tree of height h is equal to h.

Property 4: If a binary tree contains n nodes, then its maximum height possible is n and minimum height possible is $\lceil \log_2(n+1) \rceil$.

Property 5: In a non empty binary tree, if n is the total number of nodes and e is the total number of edges then $e = n-1$.

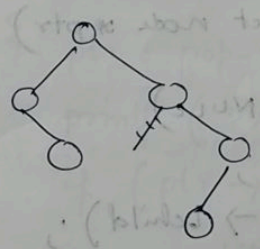
Property 6: For any non-empty binary tree, if n_0 is the number of nodes with no child and n_2 is the number of nodes with two children, then $n_0 = n_2 + 1$.

Property 7: A strictly binary tree with n nonleaf nodes has n+1 leaf nodes.

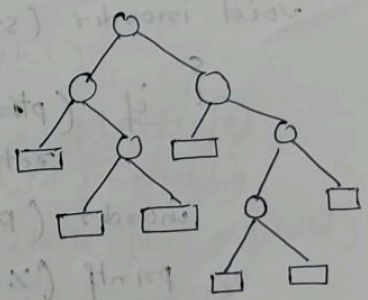
Property 8: A strictly binary tree with n leaf nodes always has 2n-1 nodes.

Property 9: In an extended binary tree, if E is the external path length, I is the internal path length and n is the number of internal nodes, then

$E = I + 2n$



Binary tree.



Extended Binary Tr

Property 10: If height of a complete binary tree is h $h \geq 1$, then the minimum number of nodes possible is 2^{h-1} and maximum number of nodes possib is $2^h - 1$.

Recursive functions for three traversals:

```
void preorder (struct node *ptr)
{ if (ptr == NULL)
        return;
  printf ("%d", ptr -> info);
  preorder (ptr -> lchild);
  preorder (ptr -> rchild);
}

void inorder (struct node *ptr)
{ if (ptr == NULL)
        return;
  inorder (ptr -> lchild);
  printf ("%d", ptr -> info);
  inorder (ptr -> rchild);
}

void postorder (struct node *ptr)
{ if (ptr == NULL)
        return;
  postorder (ptr -> lchild);
  postorder (ptr -> rchild);
  printf ("%d", ptr -> info);
}
```

9mp

Non-recursive function of Preorder traversal

```
void nrec-pre (struct node * root)
{ struct node * ptr = root;
  if (ptr == NULL)
    { printf ("Tree is empty \n");
      return;
    }
  push_stack (ptr);
  while (! stack_empty ())
    { ptr = pop_stack ();
      printf ("%d", ptr->info);
      if (ptr -> rchild != NULL)
          push_stack (ptr -> rchild);
      if (ptr -> lchild != NULL)
          push_stack (ptr -> lchild);
    }
  printf ("\n");
}
```

3rd The non-recursive function of Inorder traversal :

```
void nrec_in (struct node * root)
{
    struct node *ptr = root;
    if (ptr == NULL)
    { printf ("Tree is empty \n");
      return;
    }
    while (1)
    {   while (ptr -> lchild != NULL)
        { push_stack (ptr);
          ptr = ptr -> lchild;
        }
        while (ptr -> rchild = NULL)
        { printf ("%d", ptr -> info);
          if (stack_empty ())
              return;
          ptr = pop_stack ();
        }
        printf ("%d", ptr -> info);
        ptr = ptr -> rchild;
    }
    printf ("\n");
}
```

The non-recursive function of Postorder Traversal:

```
void nrec_post (struct node *root)
{
  struct node *q, *ptr = root;
  if (ptr == NULL)
  {
    printf ("Tree is empty\n");
    return;
  }
  q = root;
  while (1)
  { while (ptr->lchild != NULL)
    {
      push_stack (ptr);
      ptr = ptr->lchild;
    }
    while (ptr->rchild == NULL || ptr->rchild
                                  == q)
    {
      printf ("%d", ptr->info);
      q = ptr;
      if (stack_empty ())
        return;
      ptr = pop_stack ();
    }
    push_stack (ptr);
    ptr = ptr->rchild;
  }
  printf ("\n");
}
```

Imp Binary Tree Construction with inorder and preorder traversal. :

1) In preorder traversal, first node is the root node, so we get the root node by taking the first node of preorder.
Now all remaining nodes from the left and right subtrees of the root. To divide them into left and right subtrees we look ~~at~~ the inorder traversal.

2) In inorder traversal, root is in the middle, and nodes to the left of the root node are nodes of left subtree of node, and nodes to the right of the root node are nodes of right subtree of root node.

3) Now we can follow the same procedure for both left and right subtrees till we get an empty subtree or a single node in subtree.

Example : (1)

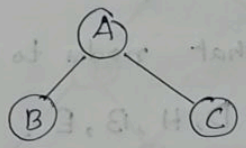
preorder : A B C

Inorder : B A C

Visit

Soln : Preorder → Root → left → Right

Inorder → Left → Root → Right



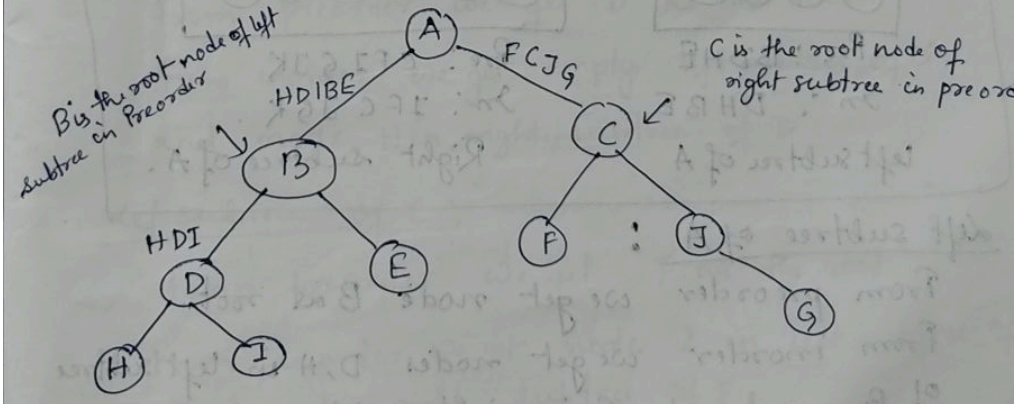
Example : (2)

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

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

Soln : Preorder → Root

Inorder → left and Right child.



Do this as (h.w) by the next given binary tree construction explanation.

Example : (3).

Preorder : ABDHE CFIGJK.

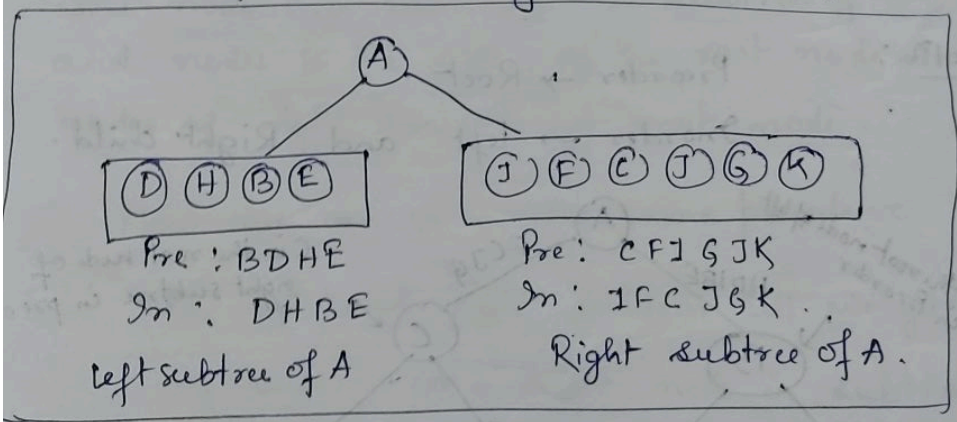
Inorder : DHBEA IFCJGK.

In preorder traversal, first node is the root node. Hence A is the root node of the binary tree.

From inorder, we see that nodes to the left of root node A are nodes D, H, B, E.

- So these nodes from the left subtree of A.

Similarly nodes I, F, C, J, G, K from the right subtree of A since they are to the right of A.



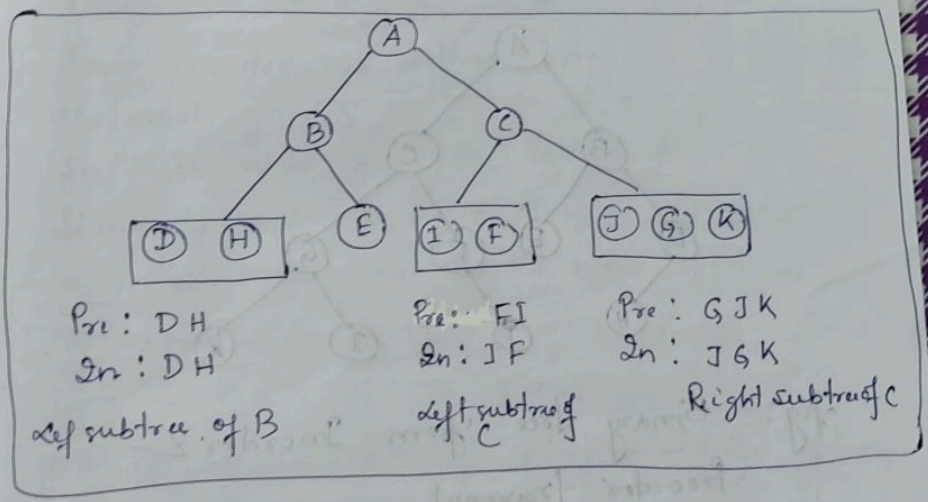
Left subtree of A :

From preorder we get node B as root.

From inorder we get nodes D, H in left subtree of B, and node E in right subtree of B.

Right subtree of A :

From preorder we get C as the root.
From inorder we get nodes I, F in left subtree of C, and nodes J, G, K in right subtree.



Lef subtree of B :

From preorder we get D as the root.
From inorder we get empty left subtree of D, and node H in right subtree of D.

Lef subtree of C :

From preorder we get F as the root.
From inorder we get node I in left subtree of F and empty right subtree of F.

Right subtree of C:

From preorder we get G as the root.
From inorder we get node J in left subtree of G and node K in right subtree of G.

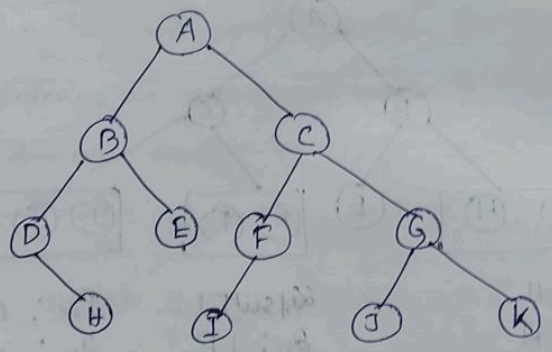


fig: Binary tree from Inorder & Preorder traversal.

—— x ——

Creation of binary tree from inorder and postorder traversals.

Same procedure with inorder and preorder traversal to construct a binary tree but the only difference is that here we get the root node by taking the last node of postorder traversal.

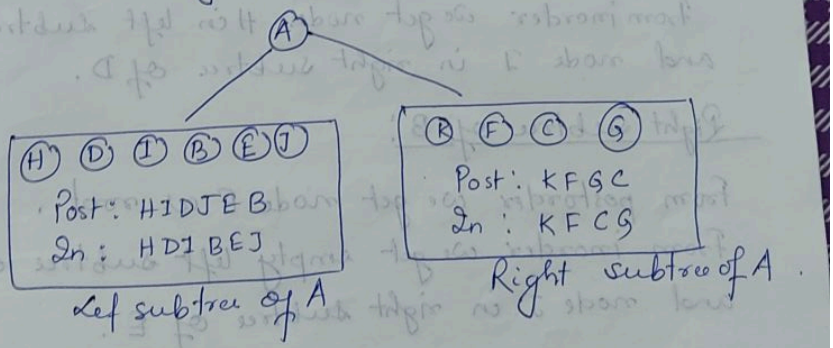
Example : (4).

Postorder : HIDJEBKFGCA

Inorder : HDIBEJAKFCG

Node A is last node in postorder traversal. So it will be the root of the tree.

From inorder, we see that nodes to the left of the root node A are nodes H, D, I, B, E, J, So these nodes from the left subtree of A. Similarly nodes K, F, C, G from the right subtree of A since they are to the right of A.



Left subtree of A

Right subtree of A.

Left subtree of A:

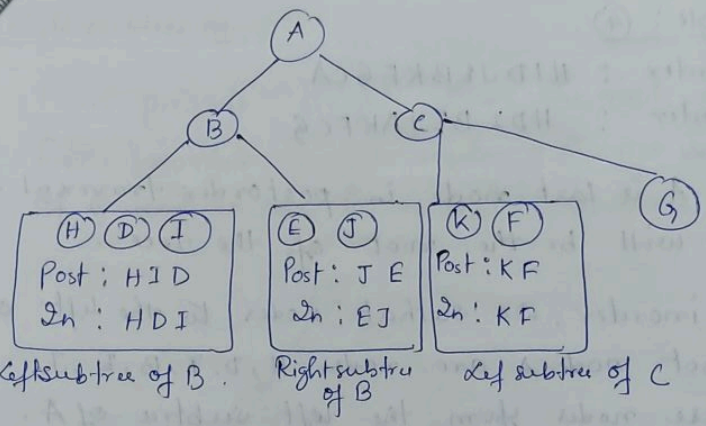
From postorder we get node B as root.

From inorder we get nodes H, D, I in left subtree of B and nodes E, J in right subtree of B.

Right subtree of A:

From postorder we get node C as root.

From inorder we get nodes K, F in left subtree of C and node G in right subtree of C.



Left subtree of B :

From postorder we get node D as root.
From inorder we get nodes H in left subtree of D, and node I in right subtree of D.

Right subtree of B :

From postorder we get node E as root.
From inorder we get empty left subtree of E, and node J in right subtree of E.

Left subtree of C :

From postorder we get node F as root.
From inorder we get node K in left subtree of F and empty right subtree of F.

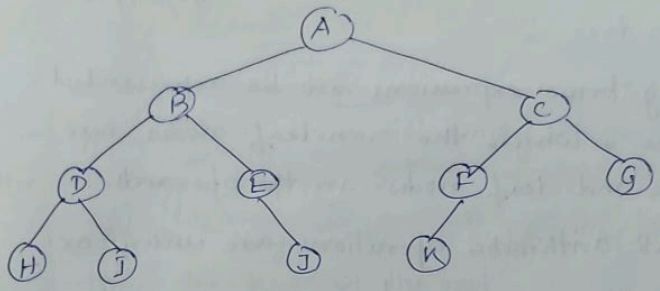


fig: Binary tree from Inorder and Postorder Traversal.

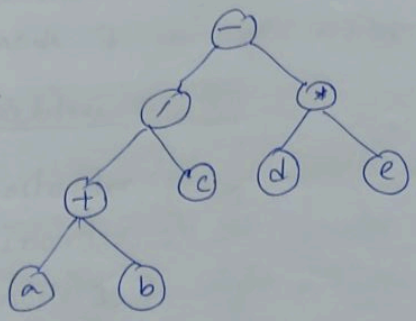
—— x ——

30/04/2020

Expression tree :

Any algebraic expressions can be represented by a tree in which the non-leaf nodes are operators and leaf nodes are the operands. Almost all arithmetic operations are unary or binary so the expression trees are generally binary trees. The left child represents the left operand while the right child represents the right operand.

Let us take an example of an algebraic expression tree.



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

Do by yourself : Draw an expression tree for the following.

(i) $(a-b*c)/(d+e/f)$

(ii) $(a+b/c)-(d+e*f)$

Write the prefix and postfix forms of the expression, by traversing the expression tree in and preorder and postorder.

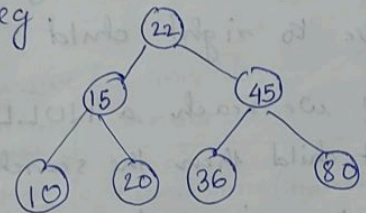
1/05/2020

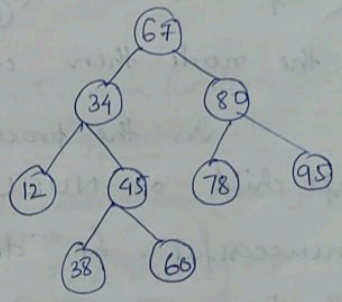
Binary search tree.

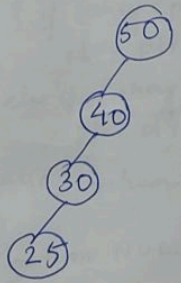
A binary search tree is a binary tree that may be empty and if it is not empty then it satisfies the following properties:

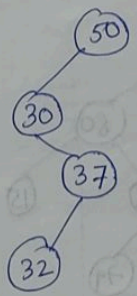
1. All the keys in the left subtree of root are less than the key in the root.
2. All the keys in the right subtree are greater than the key in the root.
3. Left and right subtrees of root are also binary search tree.

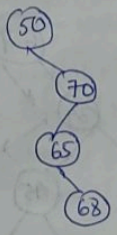
eg:

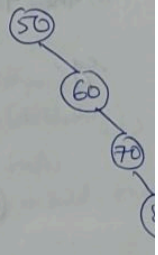












In all these trees we can see that for each node N in the tree, all the keys in left subtree of node N are smaller than key of node N and all the keys in right subtree of node N are greater than the key of node N

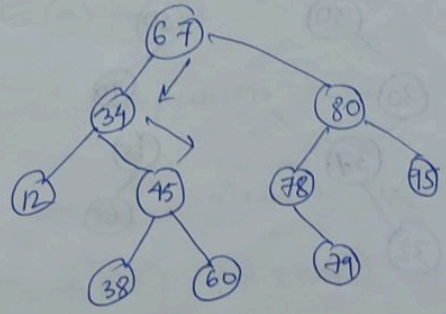
Searching in a Binary Search tree.

We start at the root node and move down the tree, and while descending when we encounter a node, we compare the desired key with the key of that node and take appropriate action.

1. If the desired key is equal to the key in the node then the search is successful.
2. If the desired key is less than the key of the node then we move to left child.
3. If the desired key is greater than the key of the node then we move to right child.

In the process, if we reach a NULL left child or NULL right child then the search is unsuccessful. i.e. desired key is not present in the tree.

eg.

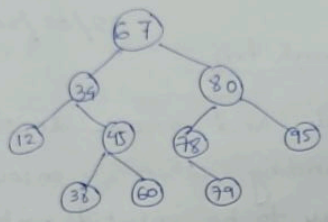


Search 45

$\because 45 < 67$, move to left child

$\because 45 > 34$, move to right child.

45 found.



Search 77

∵ 77 > 67, move to right child.

∵ 77 < 80, move to left child

∵ 77 < 78, move to left child

NULL left child

77 is not present

Recursive function for searching a node.

```
Struct node * search_rec (struct node
 * ptr, int skey)
{ while (ptr != NULL)
  { if (skey < ptr -> info)
      ptr = ptr -> lchild);
    else if (skey > ptr -> info)
      ptr = ptr -> rchild;
    else return ptr;
  }
  return NULL;
}
```

Non Recursive function for Searching a node.

```
struct node * search (struct node * ptr,
 int skey)
{ if (ptr == NULL)
    { printf ("key not found\n");
      return NULL;
    }
  else if (skey < ptr -> info)
    return search (ptr -> lchild, skey);
  else if (skey > ptr -> info)
    return search (ptr -> rchild, skey);
  else
    return ptr;
}
```

02/05/2020.

Insertion in a Binary Search tree.

We start at the root node and move down the tree and while descending when we encounter a node we compare the key to be inserted with the key of that node and take appropriate action.

1. If the key to be inserted is equal to the key in the node then there is nothing to be done as duplicate keys are not allowed.
2. If the key to be inserted is less than the key of the node then we move to left child.
3. If the key to be inserted is greater than the key of the node then we move to right child.

We insert a new key when we reach a NULL left or right child.

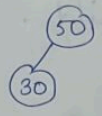
Example. Insert the keys in binary Search tree
50, 30, 60, 38, 35, 55, 22, 59, 94, 13, 98.

Step 1 Insert 50.

(50)

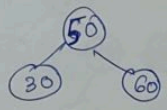
Step 2: Insert 30.

∵ 30 < 50, move to left child.



Step 3: Insert 60

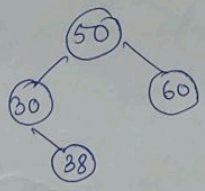
∵ 60 > 50, move to right child.



Step 4: Insert 38.

∵ 38 < 50 move to left child

∵ 38 > 30, move to right child.

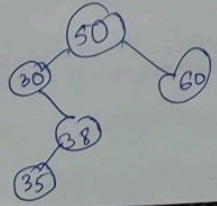


Step 5: Insert 35

∵ 35 < 50, move to left child

∵ 35 > 30, move to right child.

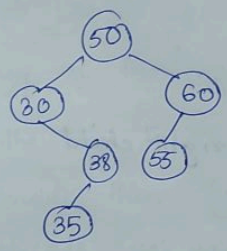
∵ 35 < 38, move to left child.



Step 6: Insert 55.

∵ 55 > 50, move to right child.

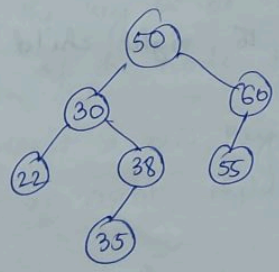
∵ 55 < 60, move to left child.



Step 7: Insert 22

∵ 22 < 50, move to left child

∵ 22 < 30, move to left child

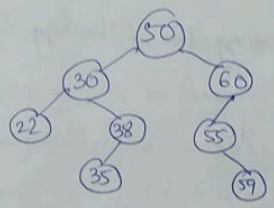


Step 8: Insert 59

∵ 59 > 50, move to right child

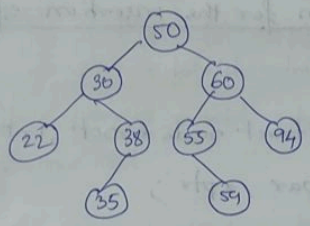
∵ 59 < 60, move to left child

∵ 59 > 55, move to right child.



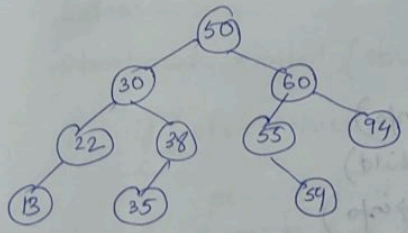
Step 9 : Insert 94.

∵ 94 > 50, move to right child
∵ 94 > 60, move to right child



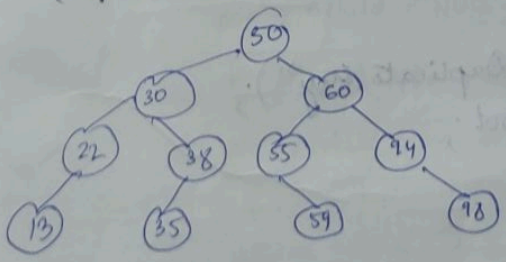
Step 10: Insert 13.

∵ 13 < 50, move to left child
∵ 13 < 30, move to left child
∵ 13 < 22, move to left child.



Step 11: Insert 98.

∵ 98 > 50, move to right child
∵ 98 > 60, move " "
∵ 98 > 94, move to " "



which is the required binary search tree.

Q. H.W:- ① Construct a BST by inserting the following data sequentially.

45 32 70 67 21 85 92 40

② 71 32 12 82 45 91 38 70 40 61

The non-recursive function for the insertion of a node is given below.

```
struct node * insert_nrec(struct node *root, int ikey)
{ struct node *tmp, *par, *ptr;
  ptr = root;
  par = NULL;
  while (ptr != NULL)
  { par = ptr;
    if (ikey < ptr->info)
        ptr = ptr->lchild)
    else if (ikey > ptr->info)
        ptr = ptr->rchild)
    else
    { printf("Duplicate key");
      return root;
    }
  }
```

```
tmp = (struct node *) malloc (size of (struct node));
tmp -> info = ikey;
tmp -> lchild = NULL;
tmp -> rchild = NULL;
if (par == NULL)
    root = tmp;
else if (ikey < par -> info)
    par -> lchild = tmp;
else    par -> rchild = tmp;
return root;
}
```

The recursive function for the insertion of a node is given below —

```
struct node *insert (struct node *ptr, int ikey)
{   if (ptr == NULL)
    {   ptr = (struct node *) malloc (size of (struct node));
        ptr -> info = ikey;
        ptr -> lchild = NULL;
        ptr -> rchild = NULL;
    }
    else if (ikey < ptr -> info)
        ptr -> lchild = insert (ptr -> lchild, ikey);
    else if (ikey > ptr -> info)
        ptr -> rchild = insert (ptr -> rchild, ikey);
    else
        printf ("Duplicate key");
        return ptr;
}
```

04/05/2020

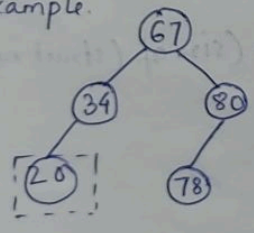
Deletion in BST.

The node to be deleted is searched in the tree. if it is found then there can be three possibilities for that node -

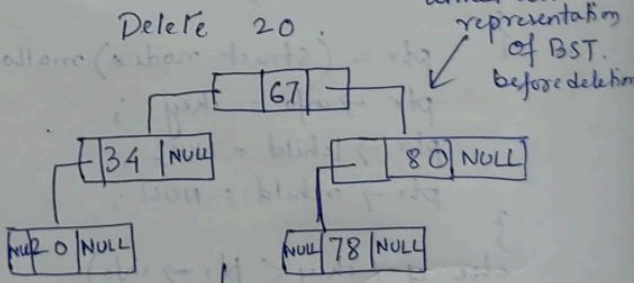
(A) Node has no child ie it is a leaf node.
(B) Node has exactly 1 child.
(C) Node has exactly 2 children.

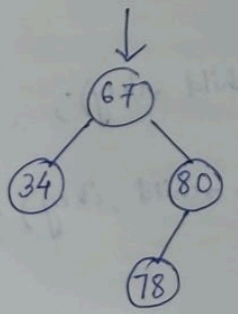
(A) **Case A :** To delete a leaf node N, the link to node N is replaced by NULL.
If the node is left child of its parent then the left link of its parent is set to NULL and if the node is right child of its parent then the right link of its parent is set to NULL. Then the node is deallocated using free().

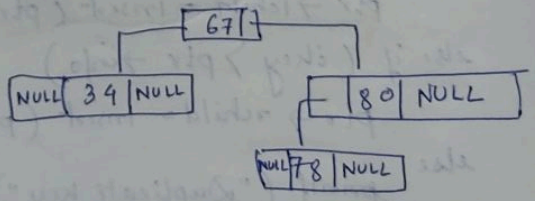
Example.



Delete 20.



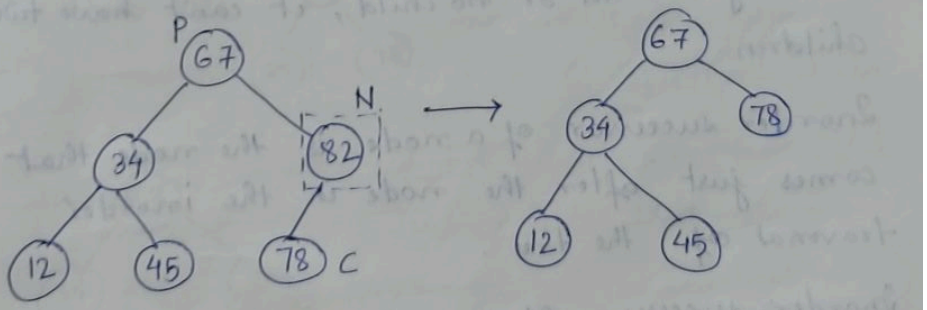


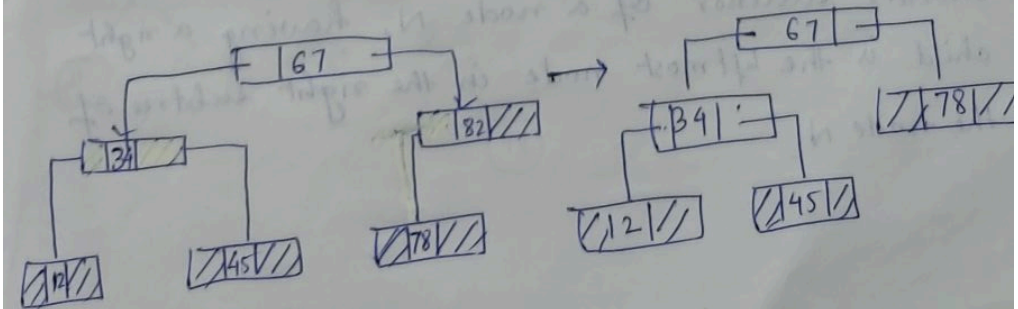


linked list representation of BST after deletion

Case B: In this case, the node to be deleted has only one child.
After deletion this single child takes the place of the deleted node
For this we just change the appropriate pointer of the parent node so that after deletion it points to the child of deleted node. After this, the node is deallocated using free().

Suppose N is the node to be deleted, P is its parent and C is its child.
If N is left child of P, then after deletion the node C becomes left child of P.
If N is right child of P, then after deletion the node C becomes right child of P.





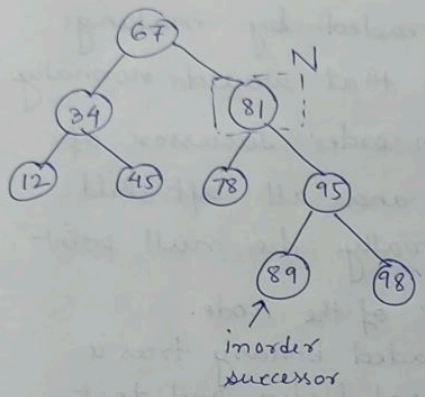
The node 82 is to deleted from the tree. Node 82 is right child of its parent 67, so the single child 78 takes the place of 82 by becoming the right child of 67.

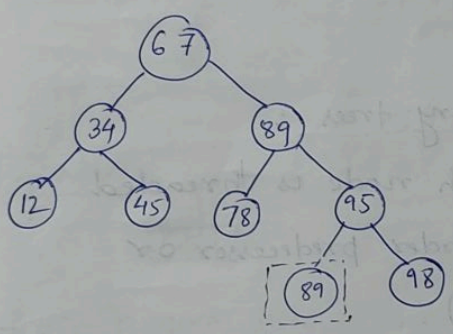
Case C : This is the case when the node to be deleted has two children.
Here we have to find the inorder successor of the node.
The data of the inorder successor is copied to the node and then the inorder successor is deleted from the tree.
Inorder successor of a node can be deleted by case A, or case B because it will have either one right child or no child, it can't have two children.

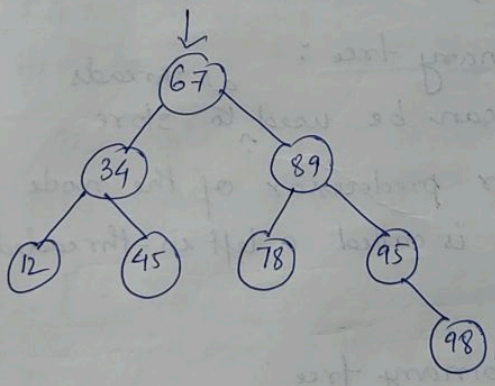
Inorder successor of a node is the node that comes just after the node in the inorder traversal of the tree.

Inorder successor of a node N, having a right child is the leftmost node in the right subtree of the node N.

example- Delete 81.







Notes

Since 81 node has two childrens, so ① we have to find out the inorder traversal of the tree.

12 34 45 67 78 81 89 95 98

Here 89 is the successor of 81

② so we replace 81 by node 89.

③ Delete inorder successor i.e. 89 using case A or case B since 89 has no child i.e leaf node.