TREE

A general tree is a finite collection of one or more element where one element is root of tree and root is disjoint into different tree and these tree are called as subtree of the tree. Subtree can be disjoint into different tree. If the tree is not disjoint, it is called leaf node.

* The max.m no. of children is called degree of the element.
* The max.m no. of its children degree is called degree of the tree.

In both the practical life and in computer memory we need to store a collection of data elements those are not linearly connected with each other. In most of the time such data elements are related with each other in a parent child relationship which makes a multilevel hierarchical structure. To represent such relationship the data organisation tree is used. There are so many examples in our daily life where we use the tree to represent the collection of information, such as family tree and organisation chart, etc.

By definition a general tree is a finite set of nodes or element where one of the element is described as the root of the tree and it presents at the top level. The other element if there are disjoined from the root and called as the subtree of tree T. These subtrees can be again disjoint into next level and so on. There are some term related with the tree. These are -

Leaf and non leaf node :- A node present in a tree having no disjoint (no child) is called as leaf node while the nodes having any child node are called as non leaf nodes. Leaf node is represented by □ and non-leaf is O

Edge :- When we depict a tree structure the line connected the child node to its parent node is called as edge.

Level :- Since tree is a hierarchical structure all the nodes of the tree must be present at a particular level. The level of the root is specified as one.

* Height of the tree :- The maxm ~~the~~ ~~ele~~ level of node is the height of the tree.
* Degree of element or node :- The no. of children the node is having is called as degree of element
* Degree of the tree :- The maxm the element degree is called as degree of the tree.

The best example of tree data structure in the computer storage is the file and directory structure organised by MS-DOS and some other operating system.

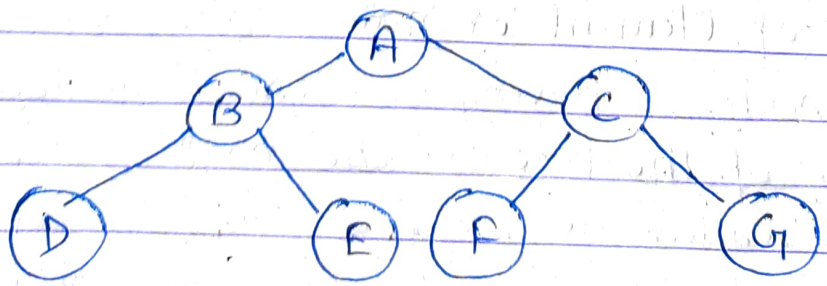
Binary tree

The binary tree is a specialized form of tree data structure. There are two major differences in between a general tree and a binary tree that is, a binary tree can be empty but a general tree cannot and another is that a node of binary tree can have a max^m. of two disjoints. Therefore it is possible to differentiate the subtree of a binary tree.

By definition we can say a binary tree is a collection of 0 or more nodes where each node if present can have children in between 0-2. The binary trees have the following properties —

(a) A binary tree T with n number of nodes has exact n-1 edge where n>0

(b) A binary tree of height h with have at least n number of nodes or almost 2^{n-1} no. of node.

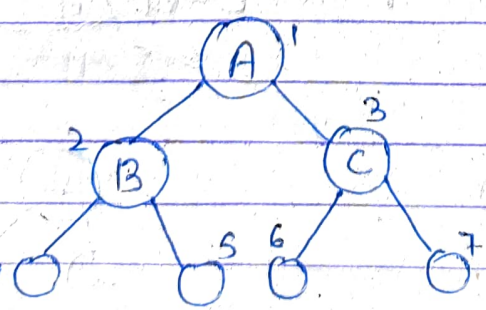


(c) A binary tree with n no. of nodes where n>0 will have height almost n and atleast $\log_2(n+1)$

(d) If each node of a binary tree is assigned with an identity no. i where i of root is ~~called~~ equal to 1 then identity no. of other nodes are evaluated by the following the given equation.

i) If a tree has n no. of nodes and the identity no. of a node i then if $2i > n$ the node is not having any left child otherwise the left child of the node is assigned with identity no. is equal to $2i$.

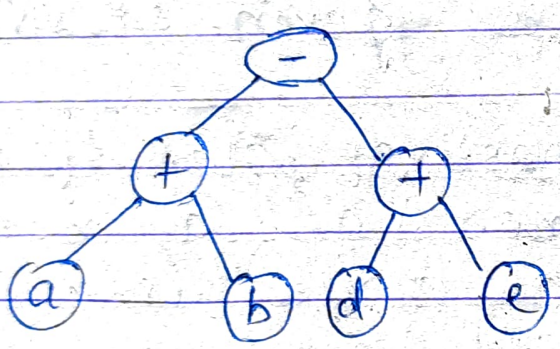
ii) If $2i+1$ is greater than n then the node will not have any right child otherwise the right child is assigned with the identity no. $2i+1$.



(iii) A binary tree having must no. of node is called a full binary tree.

(iv) If one or more node is deleted from a full binary tree but still there are nodes at the last level then the tree is called complete binary tree.

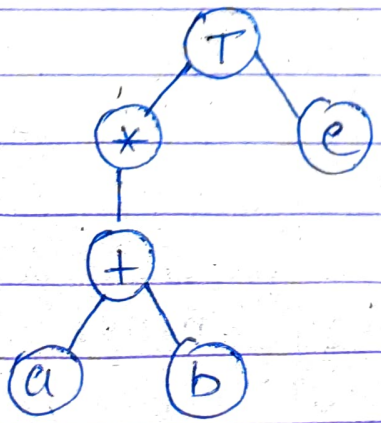
$$(a+b)-(d+e)$$
$$L \qquad R$$

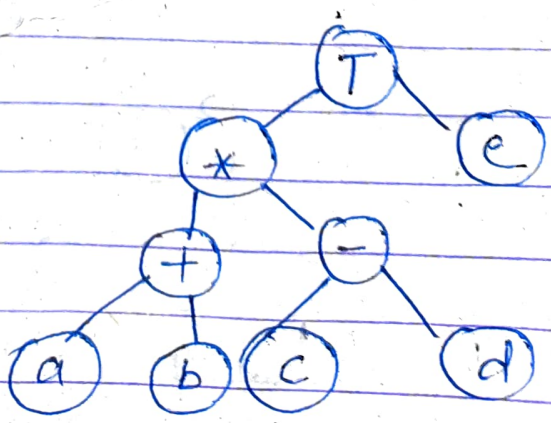


* Extended binary tree or 2-tree :-

→ If all the nodes of a binary tree have either 0 or two children then the binary tree is called as extented binary tree or 2-tree.

$$a+b*c-d/e$$





Binary search tree

A binary search tree is essentially a binary tree with two specific properties. These properties describe that :-

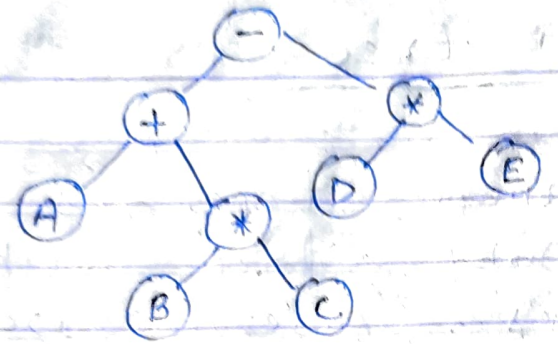
(a) A value in a binary search tree is always inserted after performing a search operation i.e a binary search tree cannot have a dublicate value.

(b) If a node is having a left child then the value of the left child is always be smaller than the node. And if there is a right child then the value of the right child will be larger than the node value.

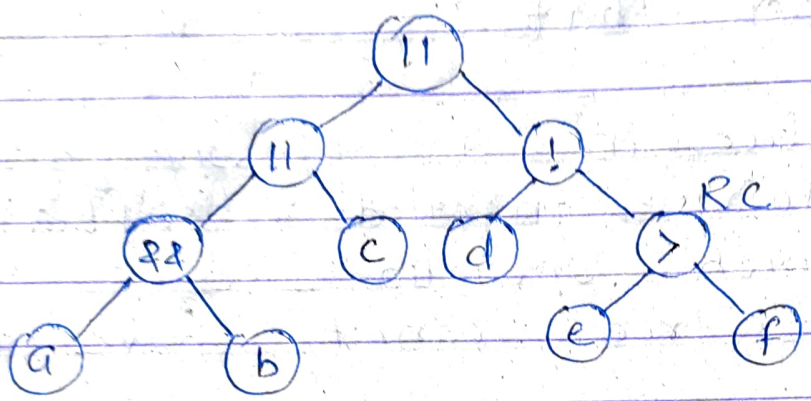
Each node of a binary search tree is also be binary search tree binary search tree.

* Preorder traverse of a Tree T.

i) Traverse Root of T.
ii) Traverse left child of Root in Preorder.
iii) Traverse Right child of Root in preorder



−+A*BC*DE.

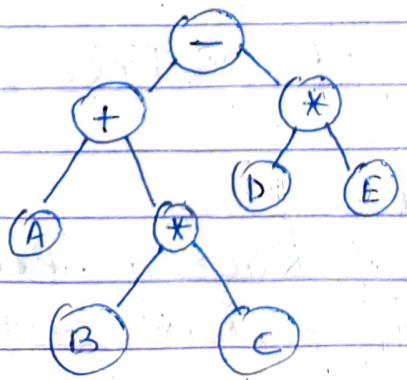


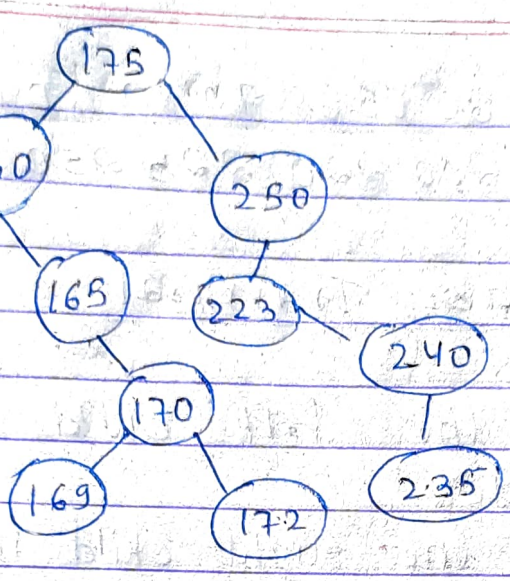
||||&&ab c!d>ef

In order Traverse of a Tree T.

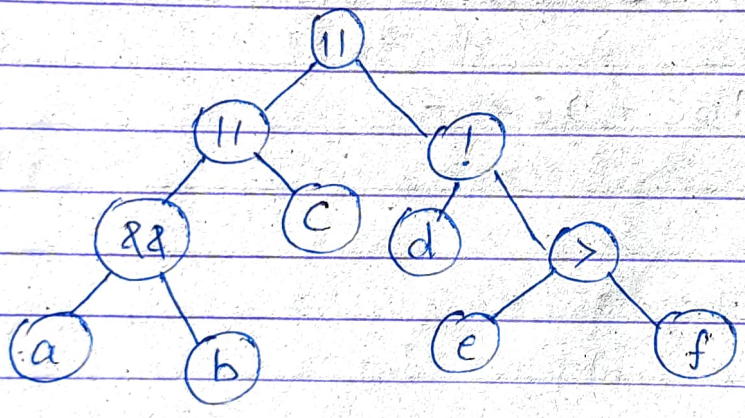
i) Traverse left child of root in inorder.
ii) Traverse root of T.
iii) Traverse Right child of root in inorder.

Inorder = **A+B*C-D*E**

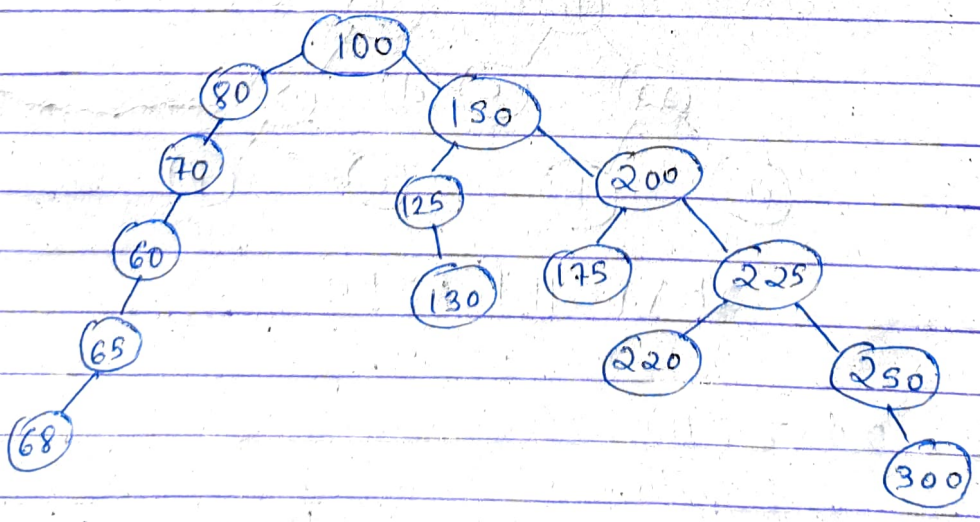




Inorder:- 160 165 169 170 172 175 223 235 240 250



a && b || c || d ! e > f



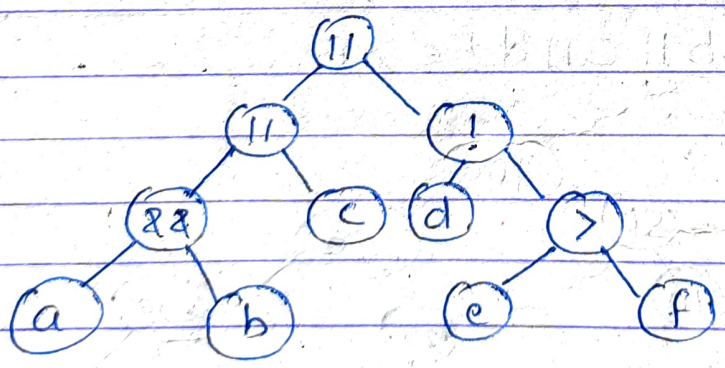
Preorder = 100 80 70 60 65 68 150 125 130 200 175 225 220 250 300.

Inorder = 60 6368 70 80 100 125 130 150 175-
200 220 225 230 300.

☆ Post order Traversal

(I) Traverse the left child in root in Post order.
II) Traverse the right child in root in post order.
III) Traverse root.

ABC + DE * -

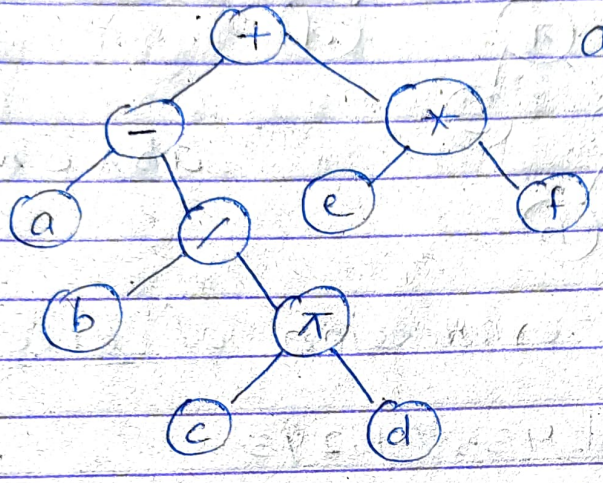


ab22c11def>!11

Q. 2007 (11)

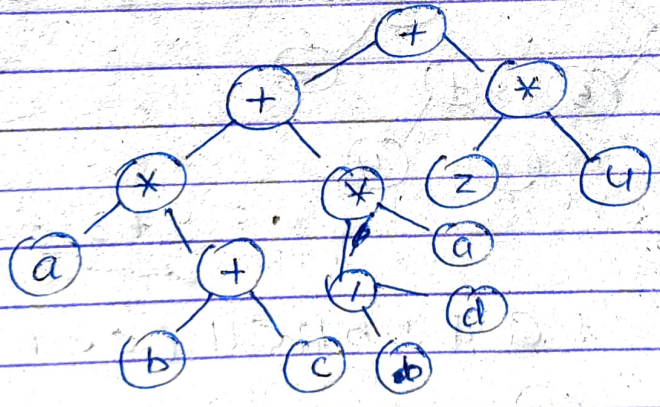
1) a-b/(c^d)+(e*f).
According to C precedence - and +

at same level with left to right associativity.



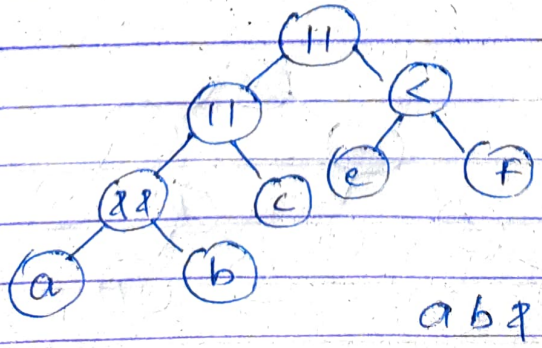
abcd^/-ef*+

ii) a * (b+c) + (b/d) * + z * u



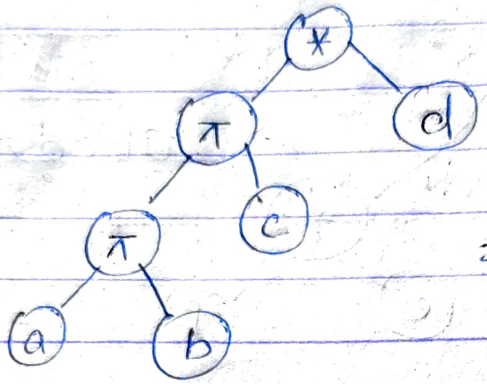
abc+*bd/a*+z4*+

iii) a && b || c || (e < f)



ab&&c||ef<||

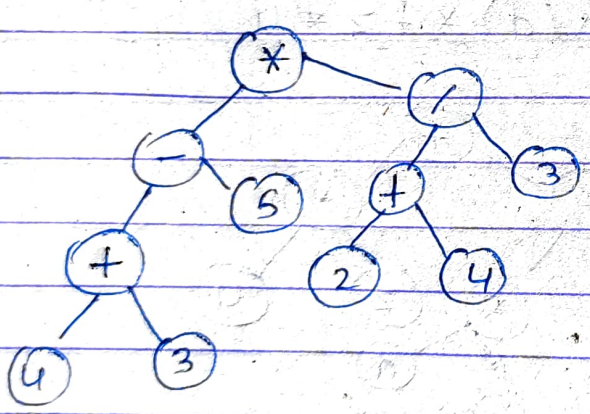
iv) a ^ b ^ c * d



= ab^c^d*

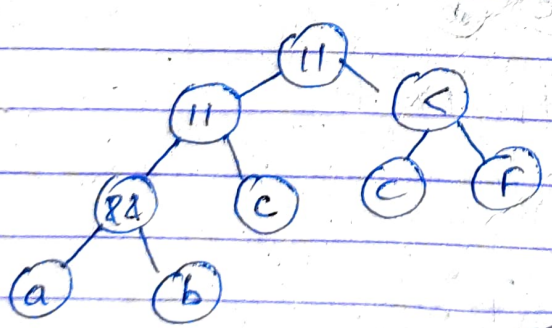
* Prefix expression of tree

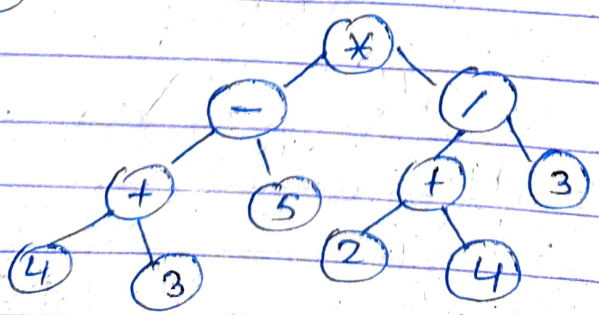
*-+435/+243



= 4+3-5*2+4/3

a && b || c || (c < f)





Traversal of binary search tree

Since a binary tree is a hierarchical structure either a with a max^m. of two disjoint therefore this structure can be transversed in three different orders. There are -

i) Preorder traversal -
ii) Postorder traversal
iii) Inorder traversal

1) **Pre-order Traversal** — In the pre-order traversal of a binary tree the root node is traversed at first then the left child of the root is traversed, if present and at last the right child of the root is traversed. All the nodes are also traversed in pre-order. This kind of traversal can be performed both by using recursive function call and by using iterative ~~algo~~ algorithm.

(A) **Recursive Pre-order traversal** — The recursive pre-order is performed by calling the function with the address of the root, which will

(**) follow the given algorithm:-

Step 1:- Start Pre-order Rec(t)
Step 2:- If T != NULL then
2.a:- Print values of T
2.b:- Call Pre-order Rec(left of T)
2.c:- " " " (Right of T)
Step 3:- End of If
Step 4:- End

```
void PreOrderRec(Tree **H)
{
  int (**H != NULL)
  {
   printf("%d", *H -> value);
   PreorderRec(*H -> LC);
   PreorderRec(*H -> RC);
  }
}

main()
 {
   Tree **H
    H = &T;
  PreorderRec(H);
  }
```

(B) Non-Recursive (iterative) Preorder Traversal :-

The iterative Preorder traversal of a binary tree is performed by creating a stack then pushing the nodes of the tree in the order required to traverse and then copying the nodes one by one and reading the value. The following algorithm used to perform such traversal.

Step 1 :- Start PreOrder (Iterative) ()
Step 2 :- Declare an array stack of tree type and an integer variable top.
Step 3 :- Set top = -1
Step 4 :- Set top = top + 1
Step 5 :- Let stack [top] = T
Step 6 :- while top >= 0
Step 7 :- Let ptr = stack [top]
Step 8 :- Let top = top - 1
Step 9 :- If ptr != NULL then
Step 9.A :- Print value of ptr.
9.B :- Let top = top + 1
9.C :- Let stack [Top] = Right to Ptr
9.D :- Let Top = Top - 1
9.E :- Let Stack [Top] = Left to ptr
Step 10 :- End of If
Step 11 :- End of while

Step 12: End of function

```
void Preorder iterative()
{
  Tree *stack[10], *ptr;
  int top = -1;
  ptr = T;
  stack[++top] = ptr;
  while (top >= 0)
  {
    ptr = stack[top--];
    if (ptr != NULL)
    {
      printf("%d", ptr->value);
      stack[++top] = ptr->RC;
      stack[++top] = ptr->LC;
    }
  }
}
```

Inorder traversal

The inorder traversal of a binary tree is performed in the way so that the left child and the right child of any node can be easily differentiated. This because in this traversal the root

node placed just in between its right and left child. In this kind of traversal the left child of the main root node is traversed first in In-order then the root node and after then the right child of the node is traverse in In-order. The In-Order traversal can be also be done in iterative mode as well as in recursive mode.

2) **Iterative or non recursive In-order traversal :—** This kind of traversal is performed by creating a stack then pushing the nodes in the manner so that the left child would be open first then the root and at last the right child. The algorithm of the function is given below:—

Step 1 :- Start InOrder Iterative
Step 2 :- Create a stack of any size and an integer variable top.
Step 3 :- Set top = -1
Step 4 :- while top1 = -1 or T1 = NULL
Step 5 :- if ptr != NULL then
S.A :- Let top = top + 1

S.B :- let stack of top = T
S.C :- let T = LC(T)

Step 6 :- Else
6.A :- let T = stack of Top
6.B :- Let Top = top - 1
6.C :- Read value of T
6.D :- let T = RC of T

Step 7 :- End of If
Step 8 :- End of while
Step 9 :- End

Post order traversal of binary tree

Once again the post-order traversal is performed by pushing and poping nodes to and from a stack but this traversal is a little bit complex. In its algor-ithm we need to keep track about the nodes that are pushed into the stack coming in the path from the root to its left most child and from the root to the left most child of its right child. To keep this track we need an extra stack to which we push a true value everytime in the secondary stack with a

node is pushed into the main stack.

Algorithm :-

Step 1:- Start Postorder iterative
Step 2:- Declare stack of Tree type, pointer ptr and two integer variables top and and tp and a stack FS.
Step 3:- Let ptr = T
Step 4:- Let Top = -1
Step 5:- Let TP = -1
Step 6:- Let Top = Top + 1
Step 7:- Let stack [top] = NULL
Step 8:- while ptr != NULL
Step 9:- Let top = top + 1
Step 10:- Let stack [top] = ptr.
Step 11:- Let FS [top] = 1.
Step 12:- If RC (ptr) != NULL
12.A:- Let top = top + 1
12.B:- Let stack [top] = RC[ptr]
12.C:- Let FS[top] = -1
Step 13:- End of If.
Step 14:- ~~Let tp = top~~ ptr = LC (ptr).
Step 15:- Let tp = top
Step 16:- Let Ptr = stack [top]
Step 17:- Let top = top - 1
Step 18:- while FS of TP = 1
Step 19:- Print values of Ptr.

Step 20:- Tp = top
Step 21:- ptr = stack[top]
Step 22:- End of while 2
Step 23:- End of while
Step 24:- End.

Creation of binary tree

A binary tree may be represented in the memory both as an array and as a linked list. But the array representation may require the number of elements that is too much than the no. of nodes present in the tree. This is because the nodes are kept in the element no. according to their identification no.
For example - the root node of the tree will be store in the index zero its left child in the index 1, its right child in the index 2, the left of left in the index 3 and so on. So if the tree is heavy in one side either in left or in right then the no. of element required to store the nodes will be almost double. In addition to store the tree as an

we must have to preknown about the height of tree.

Due to above restrictions the linked list is the better option to store a tree structure. To create a tree in the form of linked list at 1st we need to define the structure of node which will have atleast three parts :-

Member to hold the data items
Pointer to hold the left child address.
" " " " right " "

After defining the node structure a pointer is declared and assigned will null to represent an empty tree. It will follow the given algorithm —

Step 1 :- Start
Step 2 :- Define structure tree with member value and Pointer LC & RC
Step 3 :- Declare Points T of tree type
Step 4 :- Set T = NULL
Step 5 :- End.

finding location of the node :-

Although the search operation is done on any kind of data structure in contact with tree. It has a greater influence this is because the tree is traversed in a non-linear way and particularly in case of binary search tree the nodes are inserted at a specific location. and they cannot be ~~dubp~~ dublicate. Therefore before every ~~location~~ operation we need to perform a search operation. The find operation follows the given algorithm :-

Step 1 :- Start find item, Pointer P, pointer L.

Step 2 :- Declare pointer Ptr and Ps of tree type.

Step 3 :- ~~Set~~ If TP = NULL. then

3.A :- Set P = NULL

3.B :- Set L = NULL

3.C :- Return

Step 4 :- If Item < val[T].

4.A :- Set Ptr = LC[T].

Step 5 :- ~~Thed~~ Else

5.A:- Set Ptr = RC[T].
Step 6:- End of If
Step 7:- Set PS = T
Step 8:- while Ptr != NULL
Step 9:- If Item = val[Ptr]
9.A:- Set L = Ptr
9.B:- Set P = PS
9.C:- Return
Step 10:- End of If
Step 11:- PS = Ptr
Step 12:- If Item < val[Ptr].
12.A:- Set Ptr = LC[Ptr]
Step 13:- Else
13.A:- Set Ptr = RC[Ptr]
Step 14:- End of If
Step 15:- End of while
Step 16:- Set L = NULL
Step 17:- Set P = PS
Step 18:- Return

Inserting a node in a binary Search tree :-

Before a inserting into a binary search first we need to perform the search operation. The search operation confirms that the given value is not present in the free as well as it detects the location of

the node who will be the parent of new node.

Once we get the location of pat parent node, by comparing the value of parent node with the new value we connect the node either to its left side or to its Right side. The insert operation follows the given algorithm:-

Step 1:- Start insert (v)
Step 2:- Declare pointer ptr, par, loc of tree type.
Step 3:- Call find (v, add of par, add. of loc.)
Step 4:- If LOC != NULL then
4.A:- Print Item already present
4.B:- Return
Step 5:- End of If
Step 6:- Set allocate memory of ptr.
Step 7:- Let val of ptr = V
Step 8:- Set RC of ptr = NULL
Step 9:- Set LC of ptr = NULL
Step 10:- If Par = NULL then
10.A:- Set T = ptr.
Step 11:- Else if val of ptr < V then
11.A:- Set RC of Par = ptr
Step 12:- Else

12. A:- Set LC of par = ptr
Step 13:- End of If
Step 14:- End of If
Step 15:- End.

Delete from binary search tree

Since there is no fixed point to delete from within a tree and we cannot refer a data item randomly. We have to perform deletion from a binary search tree by finding the location of a specified item. Therefore the delete operation preceeded by find operation which detects the location of the item to be deleted as well as location of its Parent.

If we find a node having a specified value then to delete that node we need consider that either this node is a leaf node or a non-leaf node. If the node is a non-leaf node then again we need to check that either it has only left child, or right child or both the child, so that its child could

could be attached to its parent in a appropriate way.

Algorithm :-

Step 1 :- start delete (Item)
Step 2 :- Declare pointer Par, loc of tree type.
Step 3 :- If T = NULL then
3.A :- Printf tree empty
3.B :- return.
Step 4 :- End of If
Step 5 :- call find (Item, add. of Par, add. of loc)
Step 6 :- If LOC = NULL then
6.A :- Print Item not present
6.B :- Return
Step 7 :- End of If
Step 8 :- If ~~loc~~ LC of loc = NULL and RC of loc = NULL then call case A (Par, loc)
Step 9 :- End of If
Step 10 :- If LC of loc ! = NULL and RC of loc ! = NULL then
10.A :- call case B (Par, loc)
Step 11 :- End of If
Step 12 :- If LC of loc = NULL and RC of loc ! = NULL then
12.A :- call case B (Par, loc)

Step 13:- End If.
Step 14:- If LC of loc != NULL and RC of loc != NULL then
14.A:- Call Case C (Par, loc)
Step 15:- End of If
Step 16:- Deallocate loc
Step 17:- End

Case-A

Step-1:- Start Case A (Par, loc)
Step-2:- If Par = NULL then
2.A:- Set T = NULL
Step-3:- Else
3.A:- If loc = LC of Par then
3.A.a.1:- Set LC of Par = NULL
3.B:- Else
3.B.b.1:- RC of Par = NULL
Step-4:- End of If
Step-5:- End of If
Step-6:- End.

Case-B

Step-1:- Start case B (Par, loc)
Step-2:- Declare pointer Ptr of tree type
Step-3:- If loc if loc != NULL then
3.A:- Ptr = LC of loc.
Step-4:- else

4.A :- Ptr ; RC of loc
Step-5 :- End of If
Step-6 :- If Par = Child then
6.A :- Set T = Ptr
Step-7 :- Else
7.A :- If loc = LC of Par then
7.B :- LC of Par = Ptr
Step-8 :- Else
8.A :- RC of Par = Ptr
Step-9 :- End of If
Step-10 :- End

Case-C

Step-1 :- Start CaseC (Par, loc)
Step-2 :- Declare pointer of tree type Ptr, PS, SUC and ParSUC.
Step-3 :- PS = loc
Step-4 :- Set Ptr = LC of loc.
Step-5 :- while LC child of Ptr != NULL
Step-6 :- Set PS = Ptr.
Step-7 :- Set Ptr = LC of Ptr.
Step-8 :- End of while
Step-9 :- SUC = Ptr
Step-10 :- ParSUC = PS.
Step-11 :- If LC of SUC = NULL and RC of SUC = NULL then
11.A :- Call case a (ParSUC, SUC)
Step-12 :- Else

Step 12.A :- Call case b (Par suc, Suc)
Step-13 :- If Par = NULL
13.A :- Set Root = Suc.
Step-14 :- Else
14.A :- If loc = LC of Par
14.A.1 :- LC of Par = Suc
14.B :- Else
14.B.1 :- Set RC of Par = Suc
14.C :- End of If
Step-15 :- End of If
Step-16 :- LC of Suc = LC of loc.
Step-17 :- RC of Suc = RC of loc.