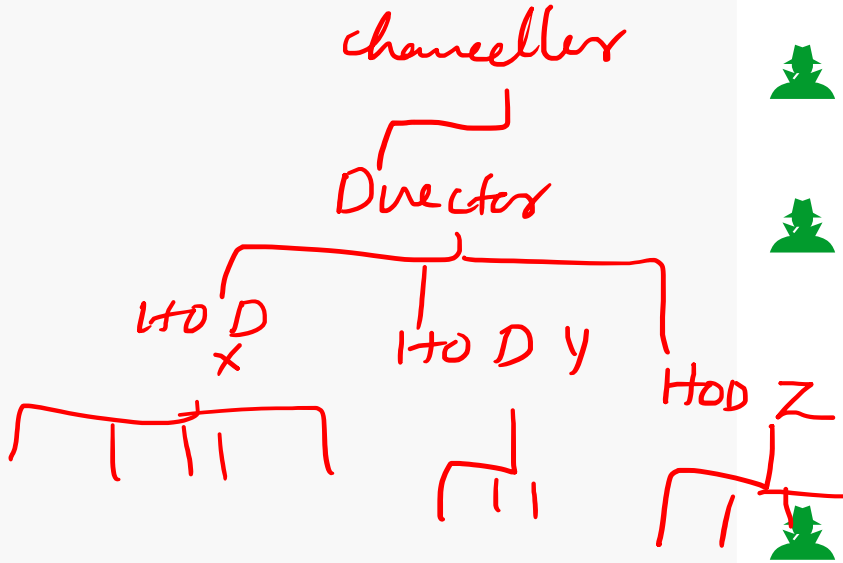

TREES.
A *non-linear*
data structure.

OBJECTIVES.

» Upon completion you will be able to:

- Understand and use basic tree terminology and concepts
- Recognize and define the basic attributes of a binary tree
- Process trees using depth-first and breadth-first traversals
- Parse expressions using a binary tree
- Understand the basic use and processing of general trees



Applications of Trees.

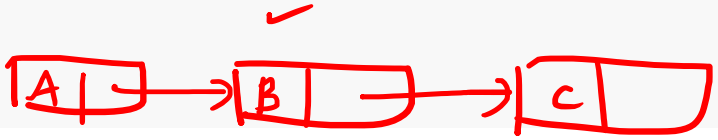
- File system
- Dynamic Spell Checking
- Network routing algorithm
- Family Tree
- Organization hierarchy

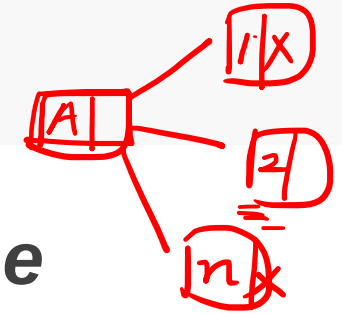
Basic Terminology.

About Trees.

LOGICAL REPRESENTATION OF TREES.

Trees



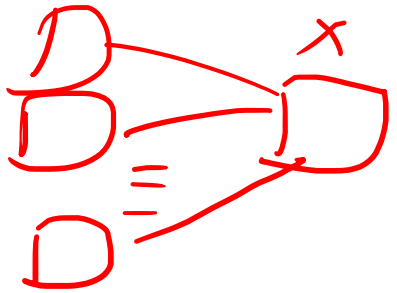


» **A non-linear list:** Each element can have ***more than one successor*** element

» Types of non-linear data structures:

Except root

1. **Tree:** An element can have only one predecessor
   - ***Two-way or binary*** (up to two successors)
   - ***Multi-way trees*** (no limitation successors)
2. **Graph:** An element can have one or more predecessors



Trees

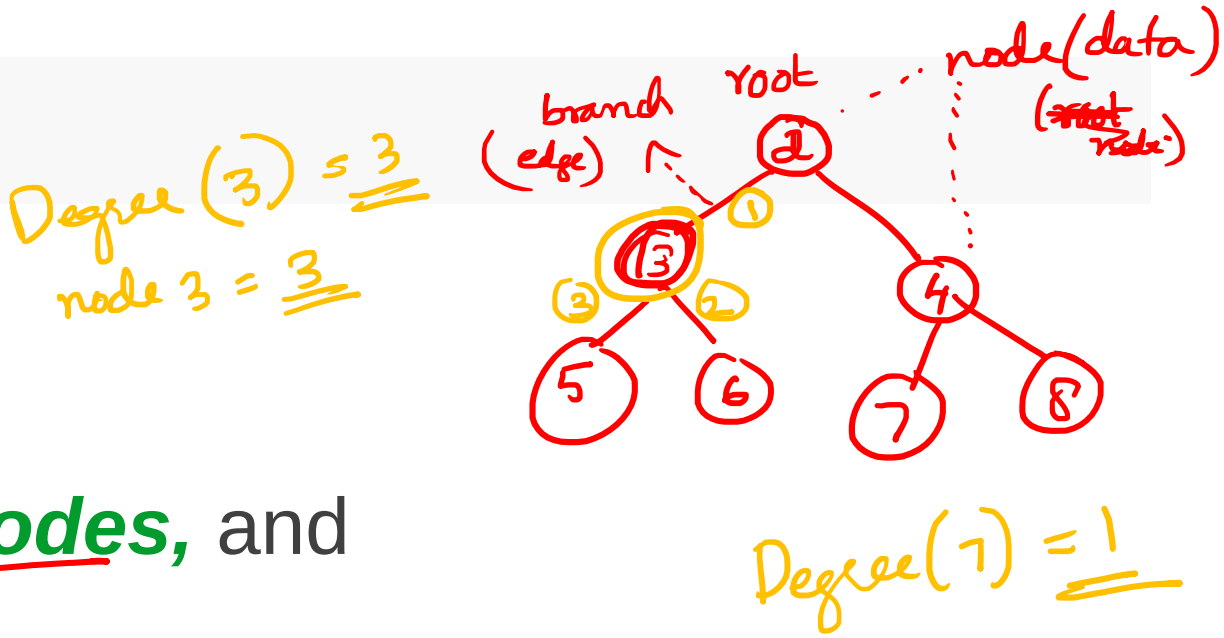
- Trees are natural structures for representing certain kinds of **hierarchical data.**(How our files get saved under hierarchical directories)

- Tree is a data structure which allows you to associate a **parent-child relationship** between various pieces of data and thus allows us to arrange our records, data and files in a hierarchical fashion.

- Trees have many uses in **computing**. For example, a *parse-tree* can represent the structure of an expression.

- **Binary Search Trees** help to order the elements in such a way that the searching takes less time as compared to other data structures.(speed advantage over other D.S)

Basic Tree concepts.



- A **tree** consists of:
  - Finite set of elements, called ***nodes,*** and
  - Finite set of directed lines called ***branches***, that connect the nodes.
- The number of branches associated with a node is the ***degree*** of the node.

- Linked list is a linear D.S and for some problems it is not possible to maintain this linear ordering.

- Using non linear D.S such as trees and graphs more complex relations can be expressed.

Basic Tree concepts.

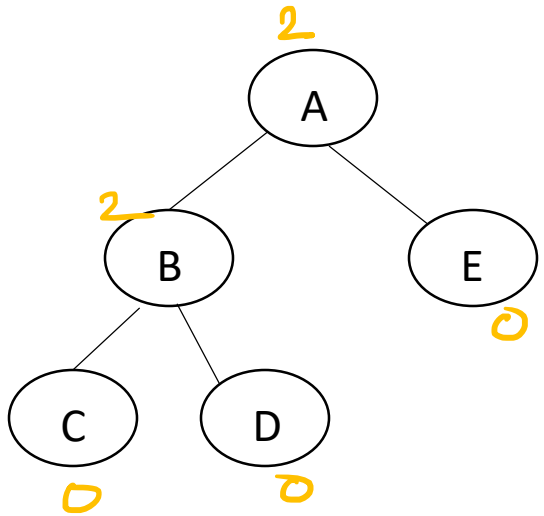
- When the branch is directed toward the node, it is ***indegree*** branch.
- When the branch is directed away from the node, it is an ***outdegree*** branch.
- The ***sum of the indegree and outdegree*** branches is the ***degree*** of the node.
- If the tree is not empty, the first node is called ***the root.***

Basic Tree concepts.

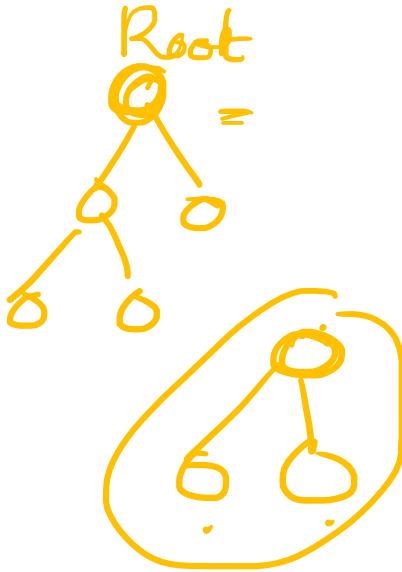
- The ***indegree of the root is***, *by definition,* ***zero***.
- With the exception of the root, **all of the nodes** in a tree must have an **indegree** of exactly **one**; that is, they may have only one predecessor.
- All nodes in the tree can have zero, one, or more branches leaving them; that is, they may have outdegree of zero, one, or more.

Indegree and Outdegree of a node:

- Indegree of B, C, D and E is 1 and that of A is 0.
- Outdegree of A and B is 2 and that of C, D and E is 0.



Directed tree:

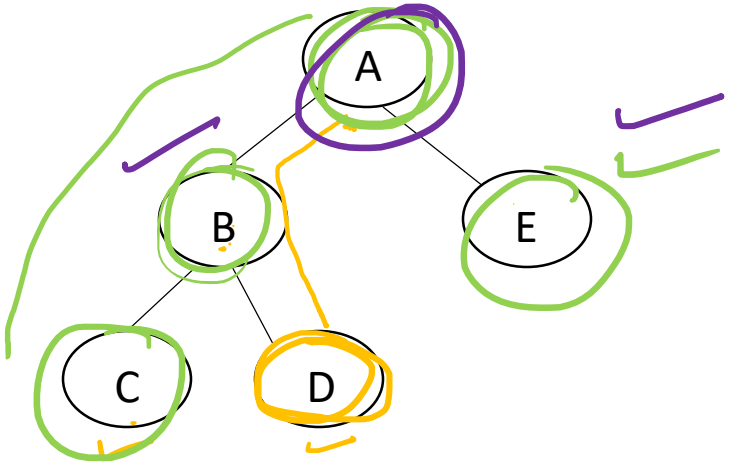


- Is a tree which has only one node with indegree 0 and all other nodes have indegree 1.
- The node with indegree 0 is called the root and all other nodes are reachable from root.

  Ex: The tree in the above diagram is a directed tree with A as the root and B, C, D and E are reachable from the root.

Ancestors and descendants of a node:

» In a tree, all the nodes that are reachable from a particular node are called the descendants of that node.



» Descendants of A are B, C, D and E.

Descendants of B are C and D.

» Nodes from which a particular node is reachable starting from root are called ancestors of that node.

» Ancestors of D and C are B and A.

Ancestors of E and B is A.

Children of a node:

» The nodes which are reachable from a particular node using only a single edge are called children of that node and this node is called the father of those nodes.

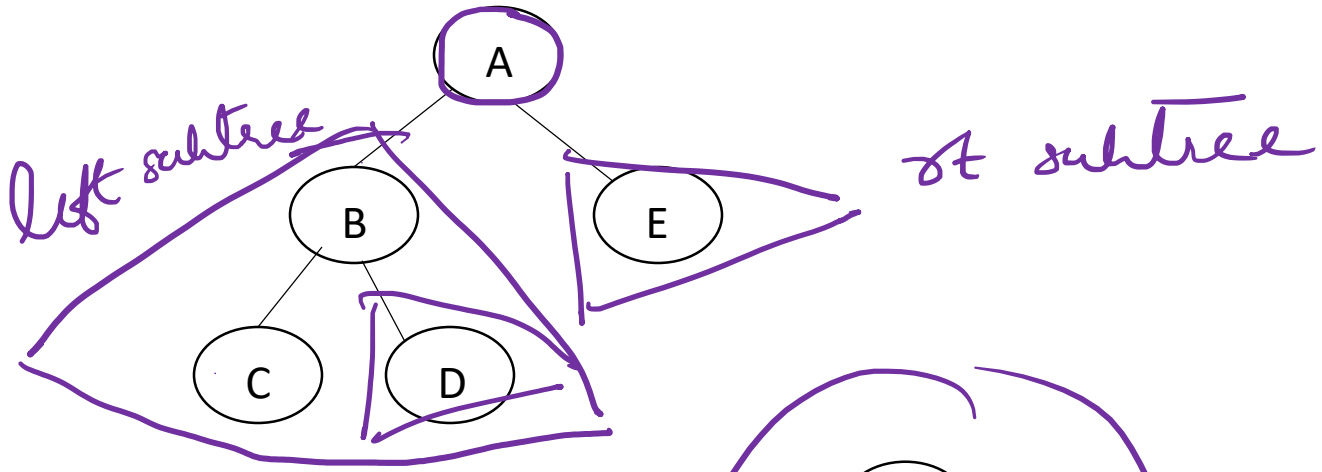
Example:

Children of A are B and E.

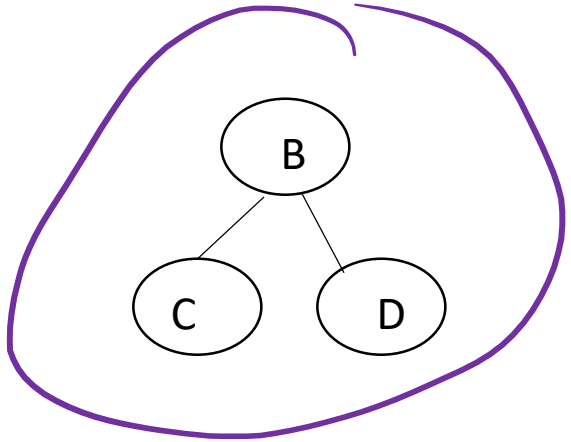
Children of B are C and D.

Left subtree and right subtree of a node:

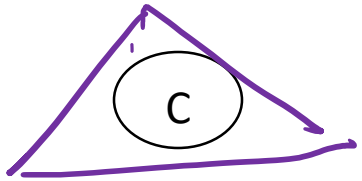
» All nodes that are all left descendants of a node form the left subtree of that node.



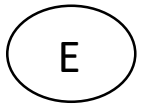
» left subtree of A is

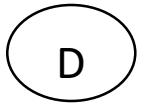


» left subtree of B is



- All nodes that are right descendants of a node form the right subtree of that node.

- right subtree of A is 

- right subtree of B is 

- right subtree of E is empty tree.

Terminal node or leaf node:

- All nodes in a tree with outdegree zero are called the terminal nodes, leaf nodes or external nodes of the tree.
- All other nodes other than leaf nodes are called non leaf nodes or internal nodes of the tree.

  In the previous tree, C, D and E are leaf nodes and A, B are non leaf nodes.

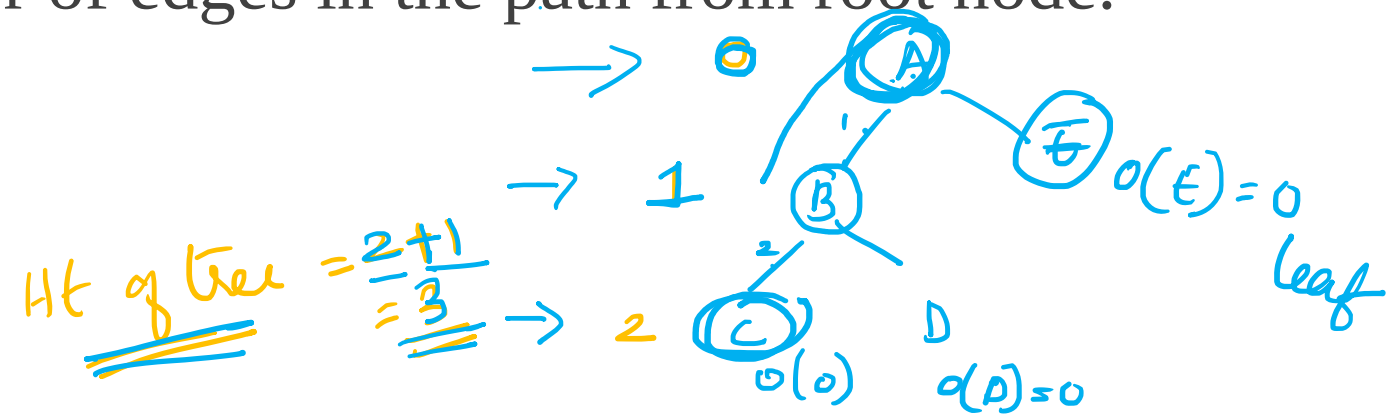
Levels of a tree:

- Level of a node is the number of edges in the path from root node.

  Level of A is 0.

  Level of B and E is 1.

  Level of C and D is 2.



Height of a tree:

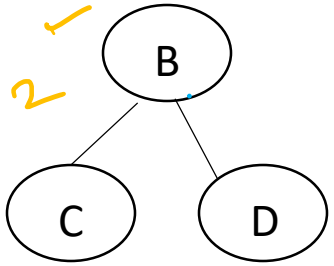
- Height of a tree is one more than the maximum level in the tree.

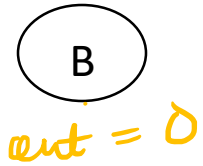
  In the previous tree , maximum level is 2 and height is 3

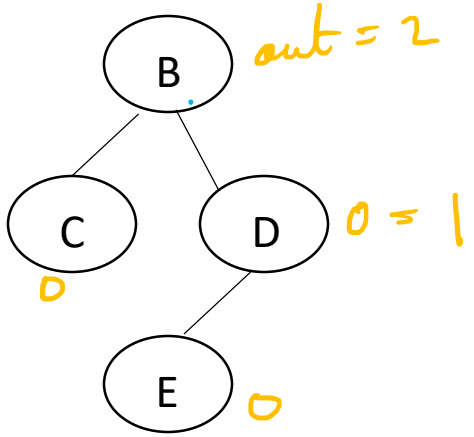
Binary trees:

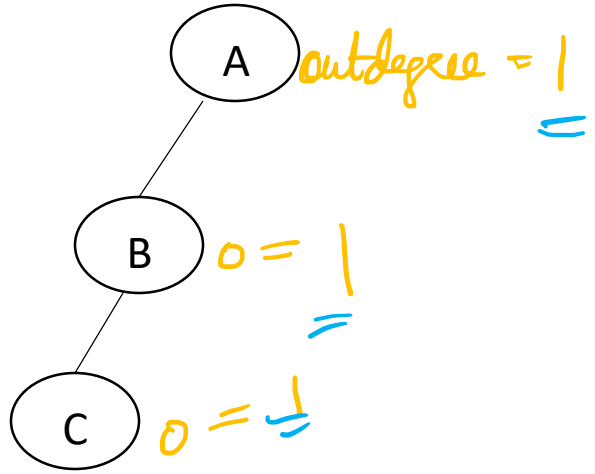
» A binary tree is a directed tree in which outdegree of each node is less than or equal to 2. i.e each node can have 0, 1 or 2 children.

Examples of binary tree:









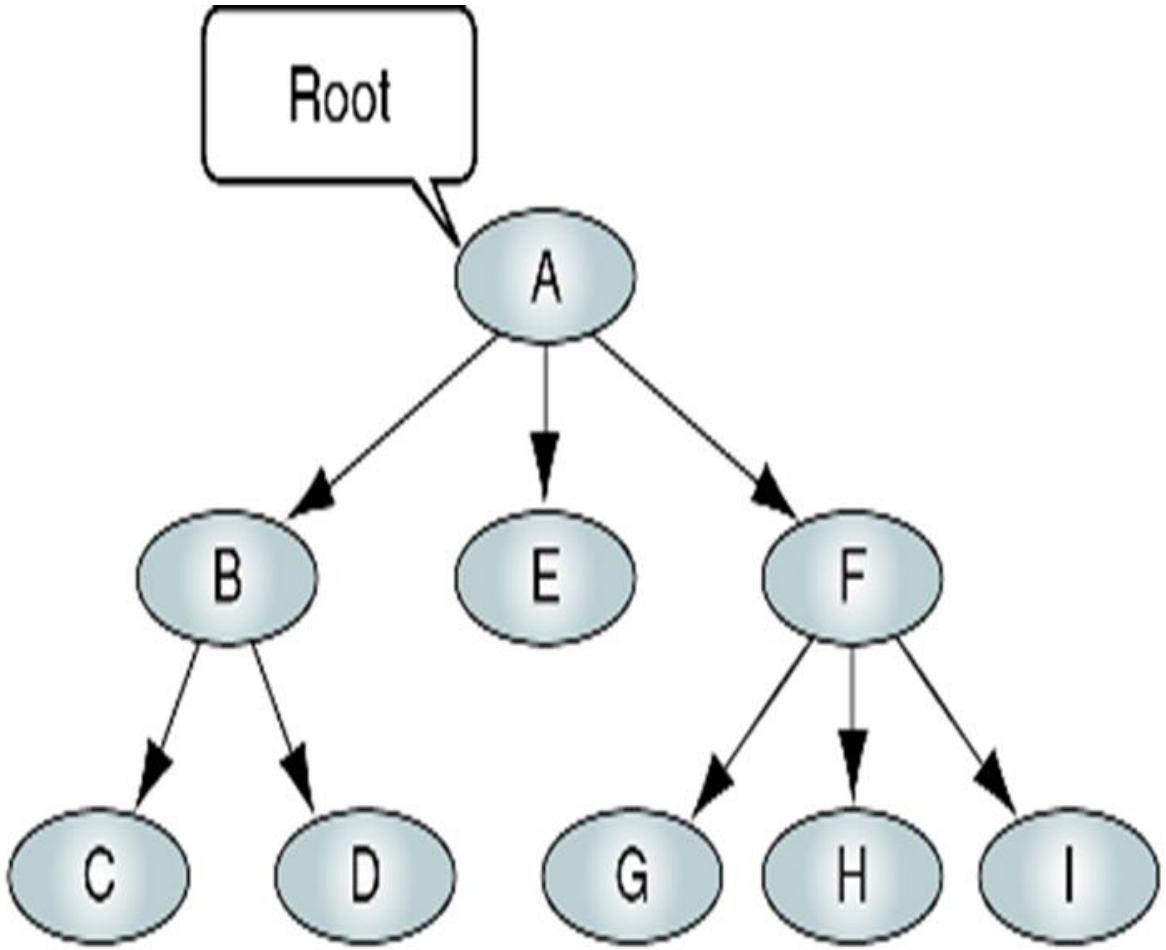
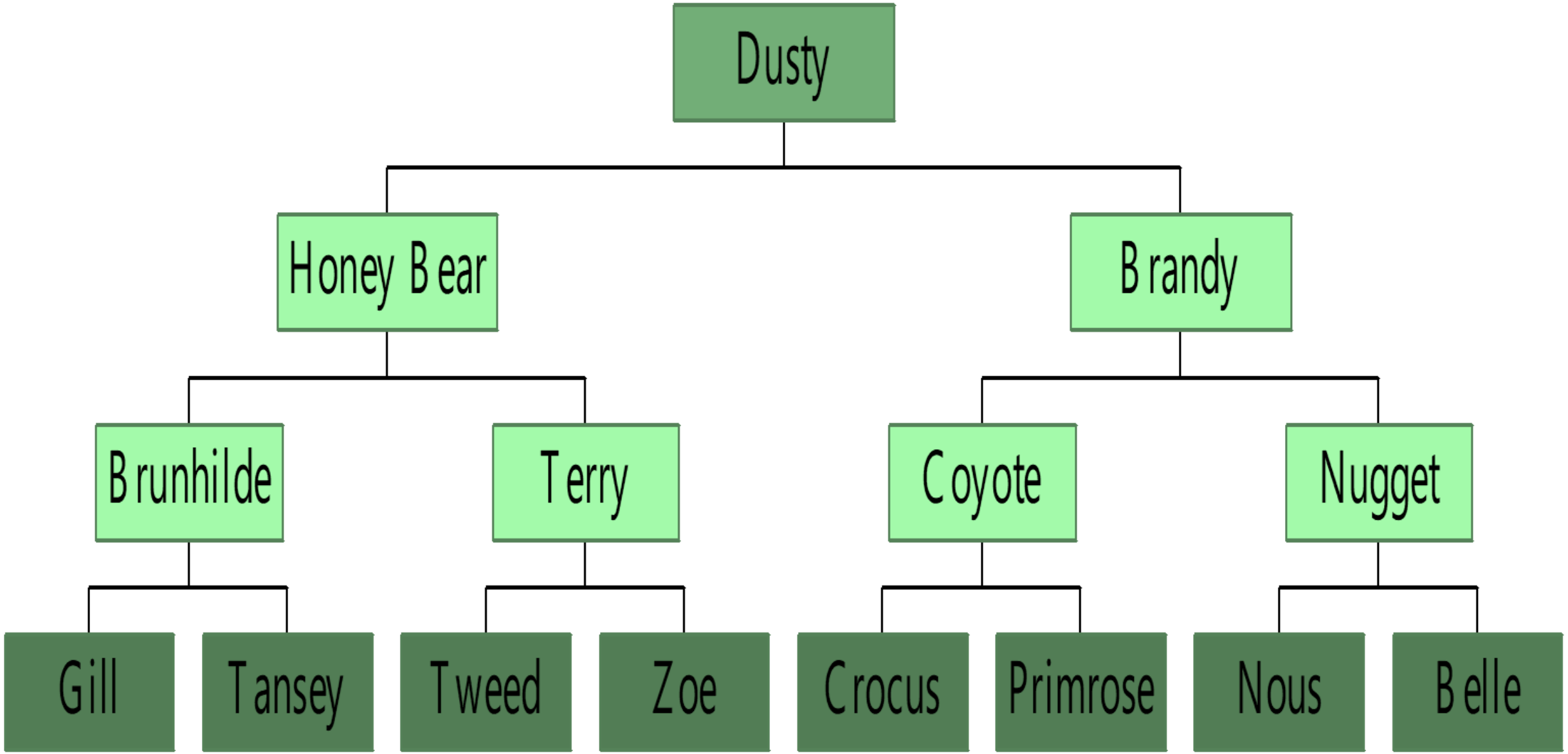


FIGURE 6-1 Tree

Tree example.


Dusty
Honey Bear
Brandy
Brunhilde
Terry
Coyote
Nugget
Gill
Tansey
Tweed
Zoe
Crocus
Primrose
Nous
Belle

Basic Tree Concepts.

- ***A leaf*** is any node with an **outdegree of zero**, that is, a node with **no successors.**
- A node that is not a root or a leaf is known as an **internal node**.
- A node is a **parent** if it has successor nodes; that is, if it has **outdegree greater than zero**.
- A node with a predecessor is called a child.

Basic Tree Concepts.

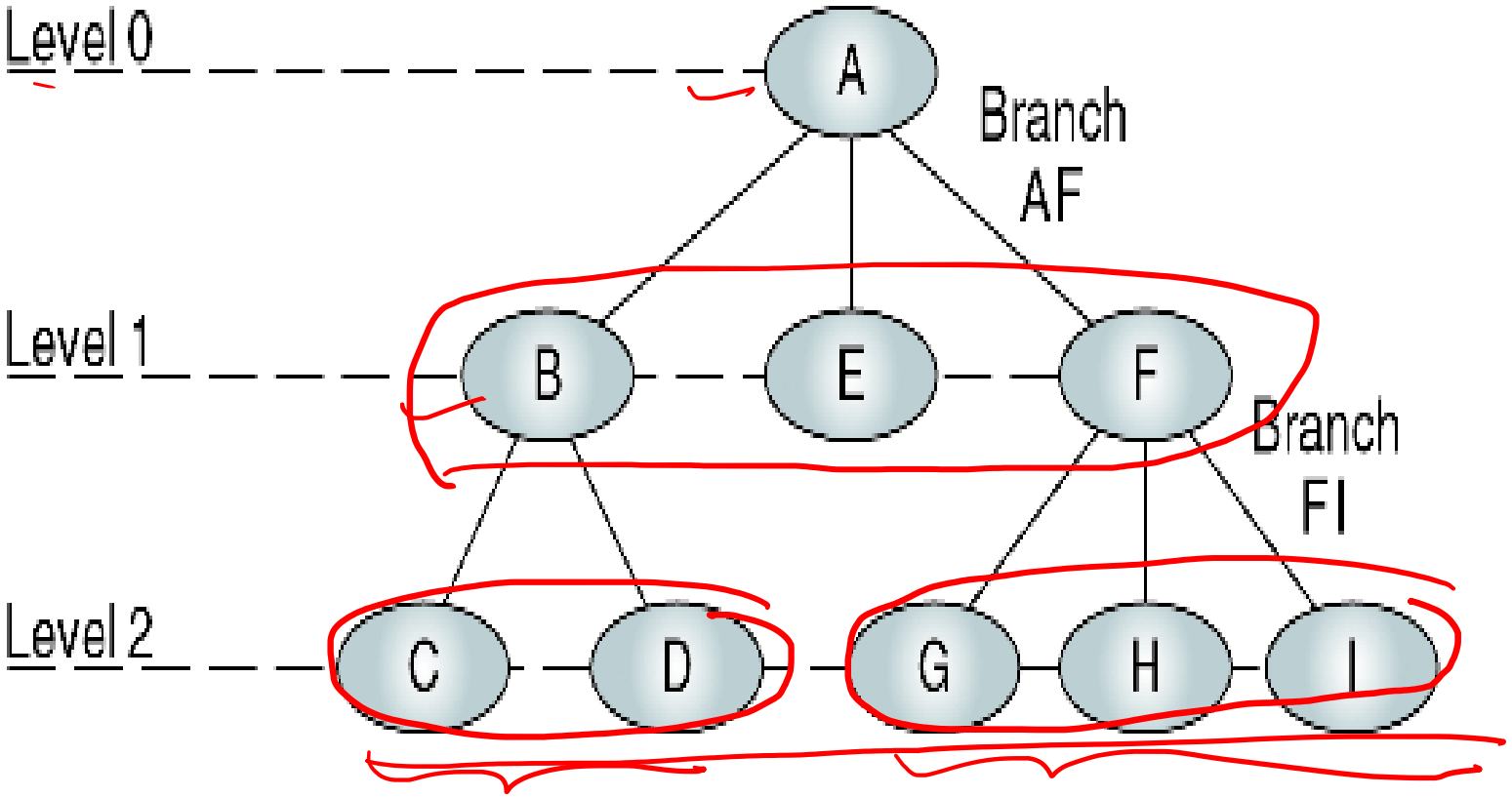
- Two or more nodes with the **same parents** are called ***siblings.***
- An **ancestor** is any node in the **path from the root to the node.**
- A **descendant** is any node in the path below the parent node; that is, all nodes in the paths from a given node to a leaf are descendants of that node.

Basic Tree Concepts.

- A **path** is a sequence of nodes in which each node is adjacent to the next node.
- The level of a node is its distance from the root. The root is at level 0, its children are at level 1, etc. …

Basic Tree Concepts.

- The **height of the tree** is the level of the leaf in the longest path from the root plus 1.
- By definition the **height** of any **empty tree is -1.**
- A **subtree** is any connected structure below the root.
- The **first node in the subtree** is known as the root of the subtree.

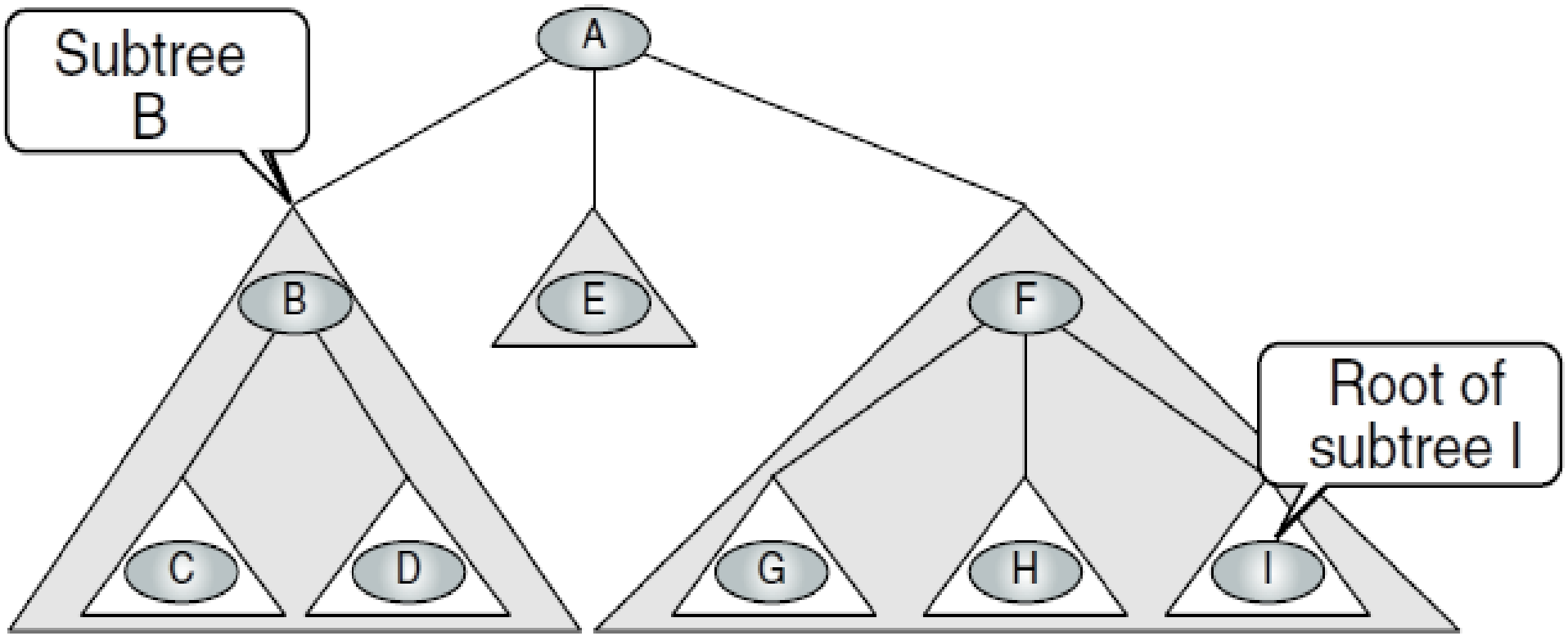


Root: A
Parents: A, B, F
Children: B, E, F, C, D, G, H, I

Siblings: {B,E,F}, {C,D}, {G,H,I}
Leaves: C,D,E,G,H,I
Internal nodes: B,F

FIGURE 6-2 Tree Nomenclature

SUB TREES.



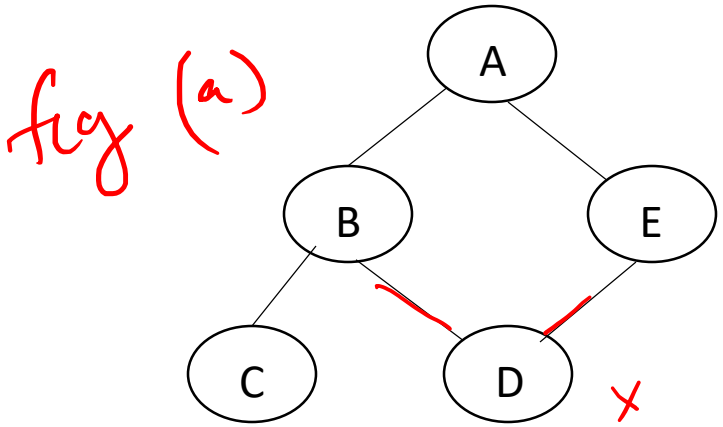
Subtrees

Binary tree

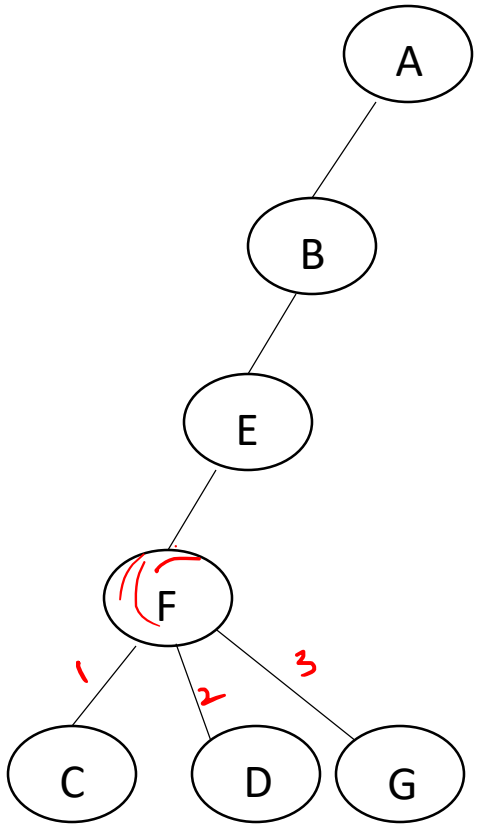
Definition:

» A binary tree is a finite set of elements that is either empty or partitioned into 3 disjoint subsets. The first subset contains root of the tree and other two subsets themselves are binary trees called left and right subtree. Each element of a binary tree is called a **NODE** of the tree.

Examples of non binary trees:



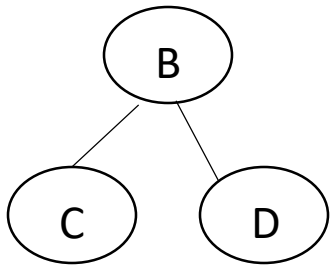
D has indegree 2, hence not directed tree

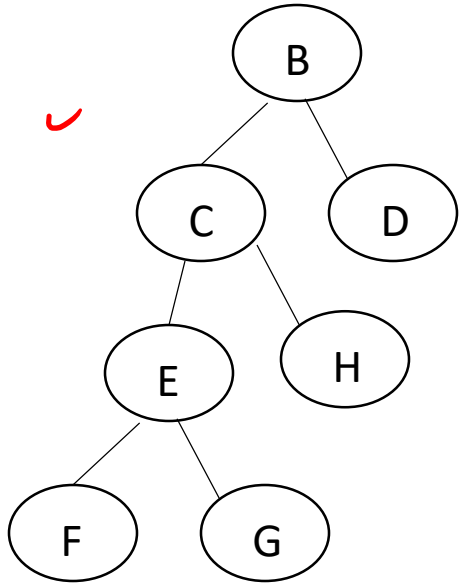


F has outdegree 3.

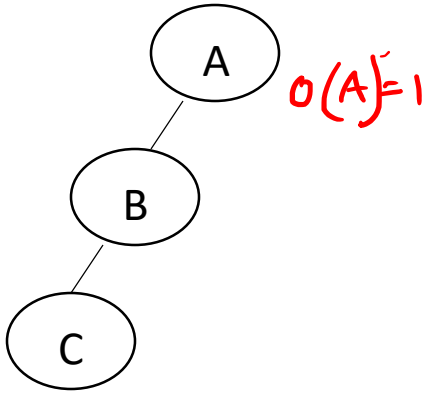
Strictly binary tree:

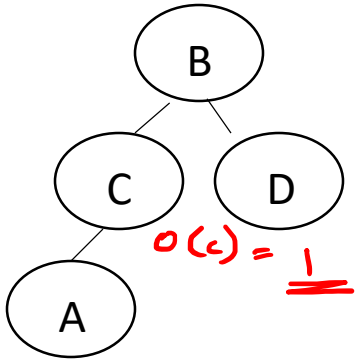
» If the out degree of every node in a tree is either 0 or 2(1 not allowed), then the tree is strictly binary tree.





Tress which are binary trees but not strictly binary:



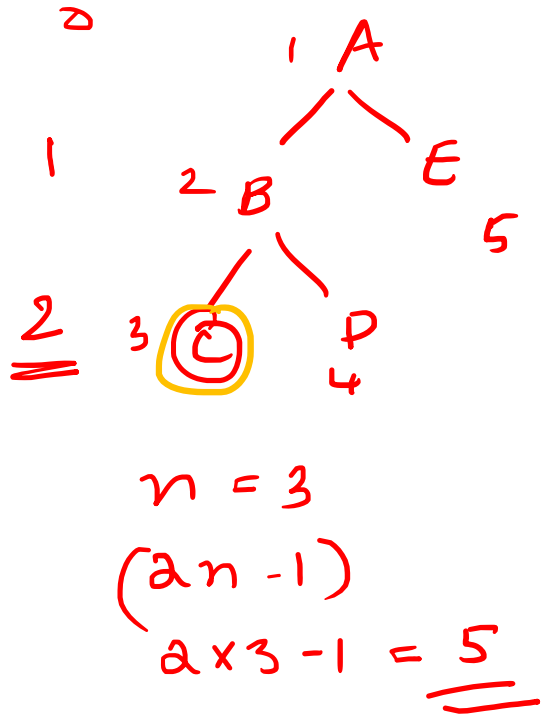


» If every non leaf node in a BT has non empty left and right subtrees ,the tree is called strictly binary tree.

Properties

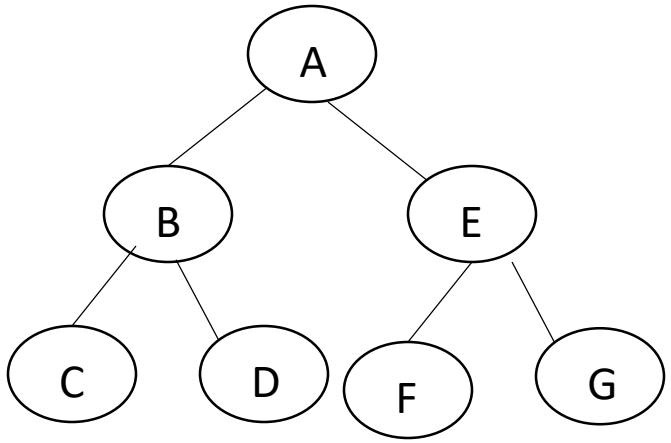
» If a SBT has n leaves then it contains 2n-1 nodes.

» **Depth:** it is the maximum level of any leaf in the tree.



Complete binary tree:

» Is a strictly binary tree in which the number of nodes at any level ‘i’ is pow(2,i).



Number of nodes at level 0(root level) is pow(2,0)→1

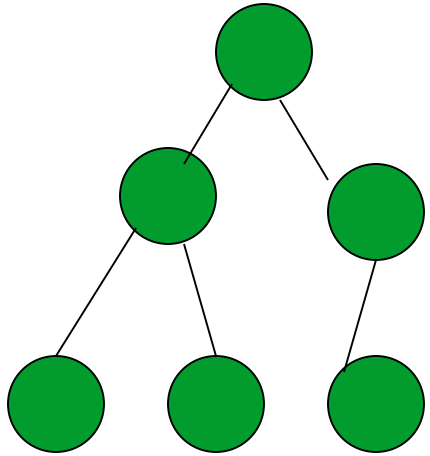
Number of nodes at level 1(B and E) is pow(2,1)→ 2

Number of nodes at level 2(C,D,F and G) is pow(2,2)→4

- A complete binary tree of depth d is the strictly binary tree all of whose leaves are at level d.
- The total number of nodes at each level between 0 and d equals the sum of nodes at each level which is equal to $\mathbf{2^{d+1} -1}$
- No of non leaf nodes in that tree $\mathbf{=2^{d}-1}$
- No of leaf nodes $\mathbf{=2^{d}}$

Almost Complete BT

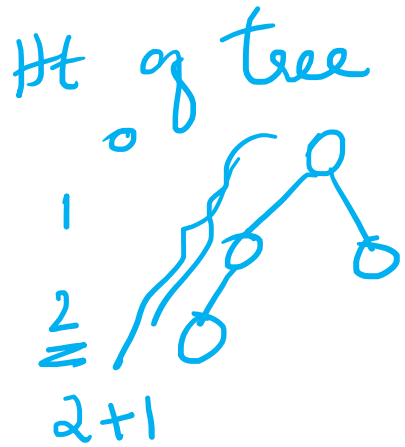
- All levels are complete except the lowest
- In the last level empty spaces are towards the right.



Some Properties of Binary Trees.

- The children of any node in a tree can be accessed by following only one branch path, the one that leads to the desired node.
- The nodes at level 1, which are children of the root, can be accessed by following only one branch; the nodes of level 2 of a tree can be accessed by following only two branches from the root, etc.
- The **balance factor** of a binary tree is the difference in height between its left and right subtrees:

$$B = H_L - H_R$$



Balance of the tree.

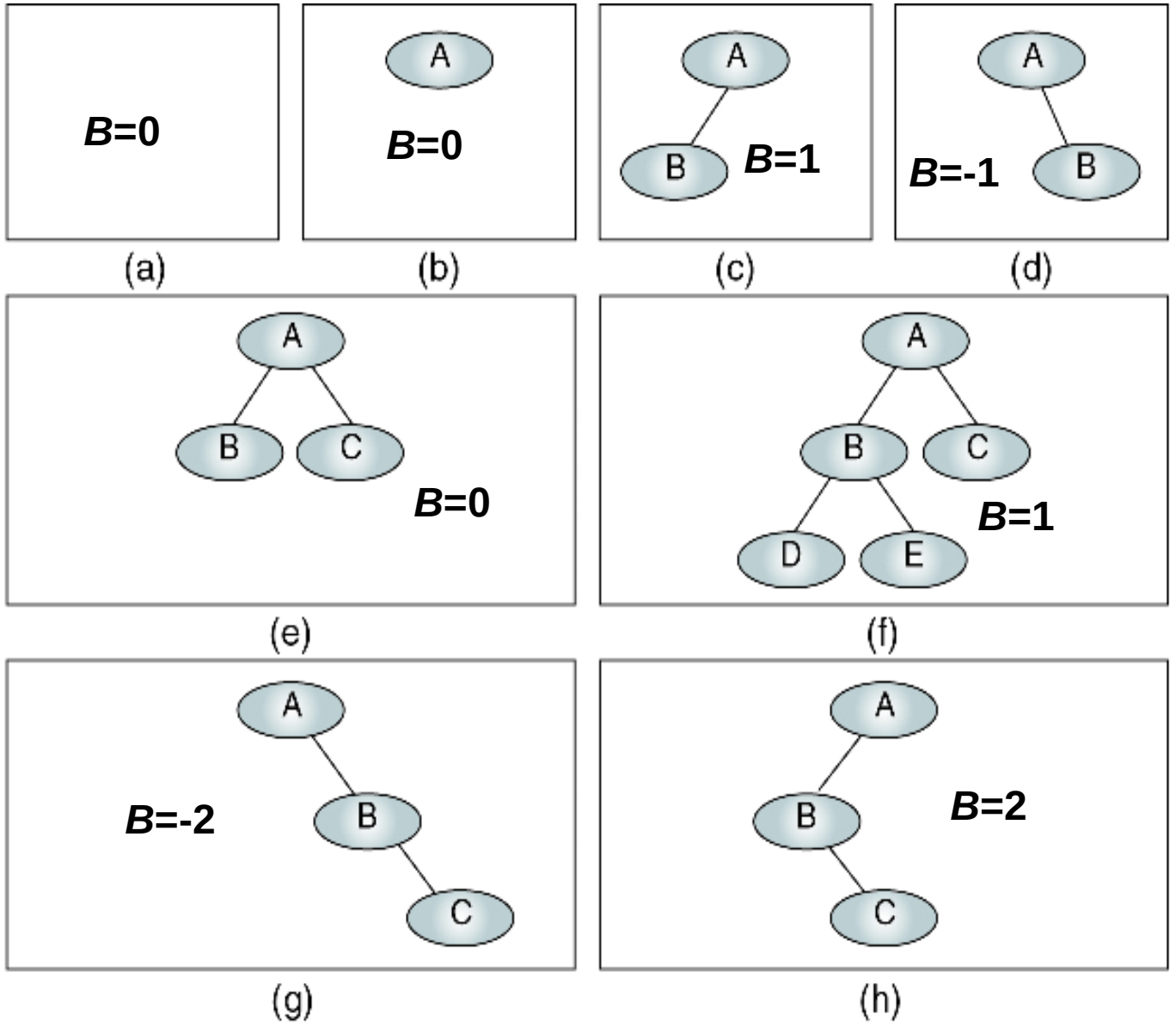


FIGURE 6-6 Collection of Binary Trees

Some Properties of Binary Trees.

- In the **balanced binary tree** (definition of Russian mathematicians Adelson-Velskii and Landis) the height of its subtrees differs by no more than one (its balance factor is -1, 0, or 1), and its subtrees are also **balanced.**

Sequential Representation

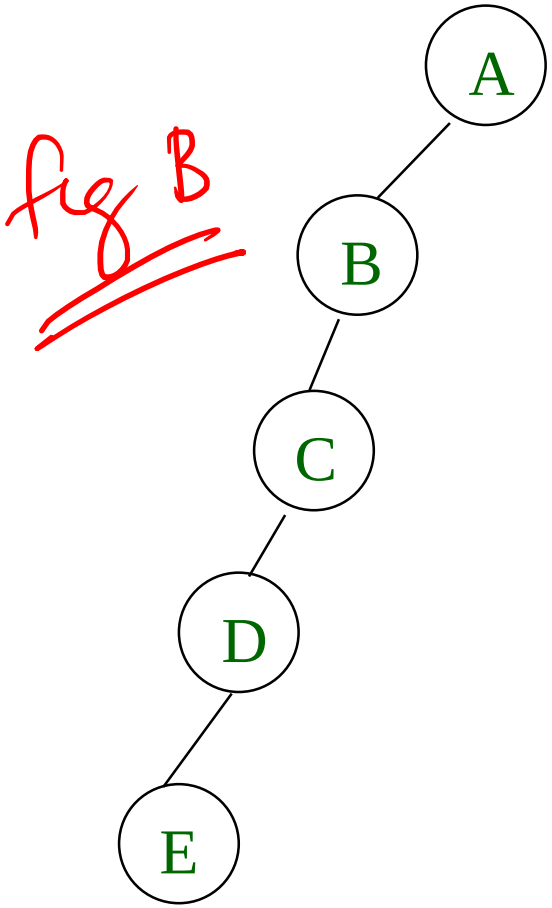
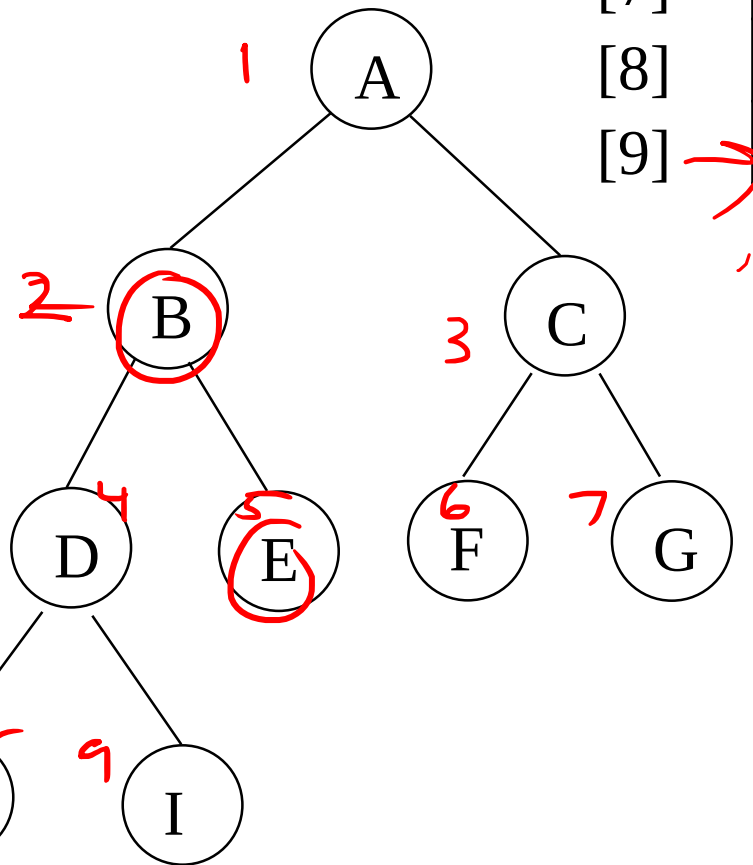


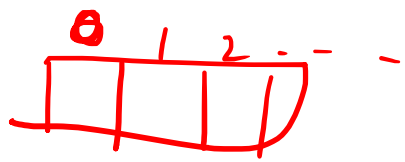
fig B

[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

(1) waste space
(2) insertion/deletion problem

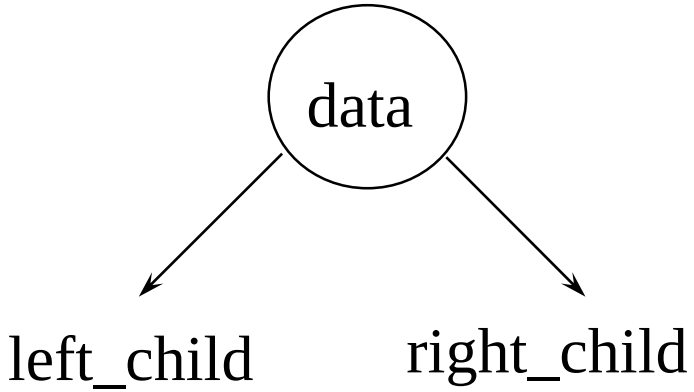
[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I





Linked Representation

```
class node {
 int data;
 node *left_child, *right_child;
};
```



Storage representation of binary trees:

» Trees can be represented using sequential allocation techniques(arrays) or by dynamically allocating memory.

» In 2nd technique, node has 3 fields

1. Info : which contains actual information.
2. llink :contains address of left subtree. //lchild
3. rlink :contains address of right subtree. //rchild

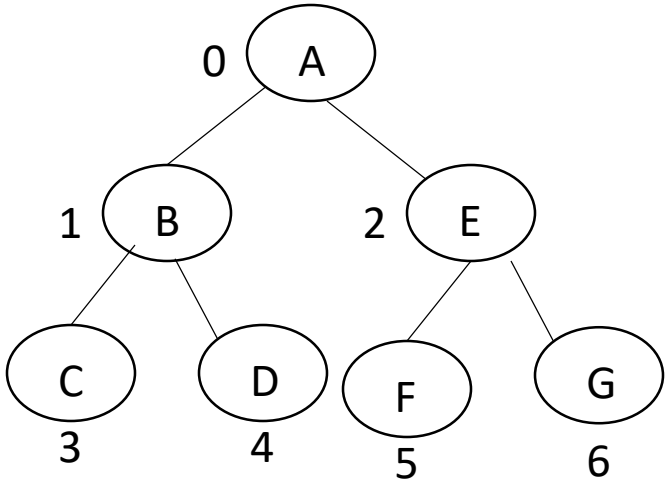
```
struct node
{
    int info;
    struct node *llink;
    struct node *rlink;
};
 typedef struct node *NODEPTR;
```

Implementing a binary tree:

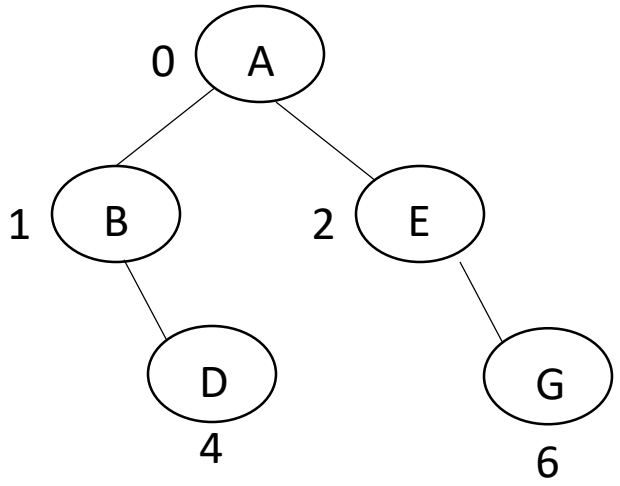
- A pointer variable root is used to point to the root node.
- Initially root is NULL, which means the tree is empty.

NODEPTR root=NULL;

Array representation of binary tree:



0	1	2	3	4	5	6
A	B	E	C	D	F	G



0	1	2	3	4	5	6
A	B	E		D		G

- Given the position ‘i’ any node, 2i+1 gives the position of left child and 2i+2 gives the position of right child.

  In the above diagram, B’s position is 1. 2*1+2 gives 4, which is the position of its right child D.

- Given the position ‘i’ of any node, (i-1)/2 gives the position of its father.

  Position of E is 2. (2-1)/2 gives 0, which is the position of its father A.

Various operations that can be performed on binary trees:

- Insertion : inserting an item into the tree.
- Traversal : visiting the nodes of the tree one by one.
- Search : search for the specified item in the tree.
- Copy : to obtain exact copy of given tree.
- Deletion : delete a node from the tree.

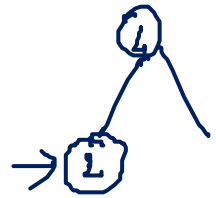
```
void insert_rec(NodePtr *root, int data) {

        if(*root==NULL) {
                NodePtr tmp = malloc(sizeof(Node));
                tmp->info= data;
                tmp->llink= tmp->rlink= NULL;

                *root = tmp;
                return;
        }

        printf("Insert to the left or right of %d?\n", (*root)->data);
        char str[10];
        scanf("%s", str);
        if(strcmp(str,"left")==0) {
                insert_rec(&((*root)->llink), data);
        } else {
                insert_rec(&((*root)->rlink), data);
        }}
```

```
void insert_iter(NodePtr *root, int data)
{
   NodePtr tmp = malloc(sizeof(Node));
        tmp->info= data;
        tmp->llink= tmp->rlink= NULL;

        if(*root==NULL)
   {
                *root = tmp;
                return;
        }

 NodePtr i; char str[10];
```

```
for(i=*root; i!=NULL; ) {
        printf("Insert to the left or right of %d?\n", i->data);
                scanf("%s", str);
                if(strcmp(str,"left")==0) {
                        if(i->llink==NULL) {
                                i->llink= tmp;
                                break;
                        } else {
                                i=i->llink;
                        }
                } else {
                        if(i->rlink==NULL) {
                                i->rlink= tmp;
                                break;
                        } else {
                                i=i->rlink;
                        }
                }}}
```

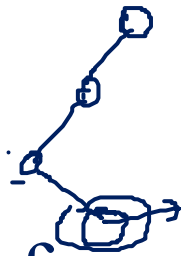
Insertion:

- A node can be inserted in any position in a tree(unless it is a binary search tree)

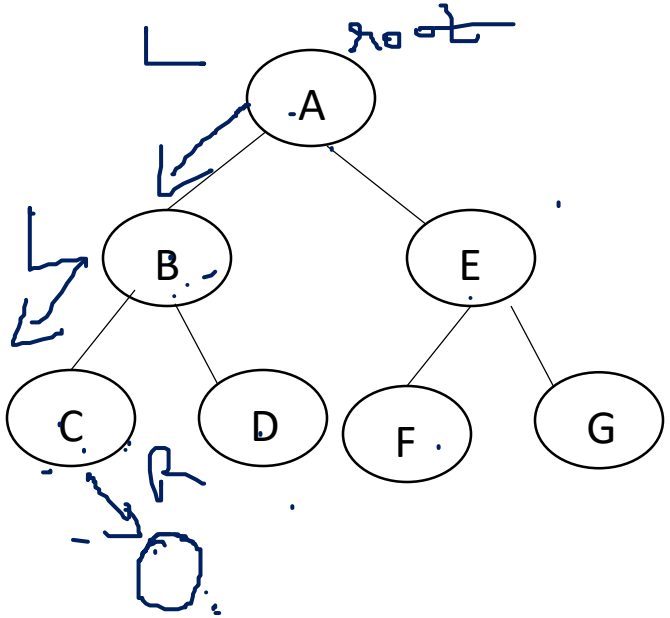


- A node cannot be inserted in the already occupied position.

- User has to specify where to insert the item. This can be done by specifying the direction in the form of a string.



- For ex: if the direction string is "LLR", it means start from root and go left(L) of it, again go left(L) of present node and finally go right(R) of current node. Insert the new node at this position.



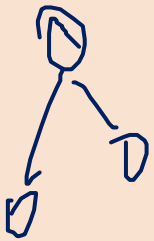
For the above tree if the direction of insertion is "LLR"

- Start from root. i.e A and go left. B is reached.
- Again go left and you will reach C.
- From C, go right and insert the node.

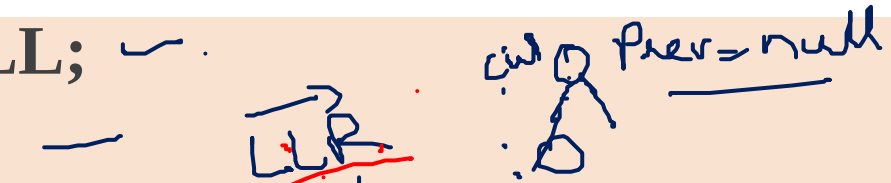
Hence the node is inserted to the right of C.

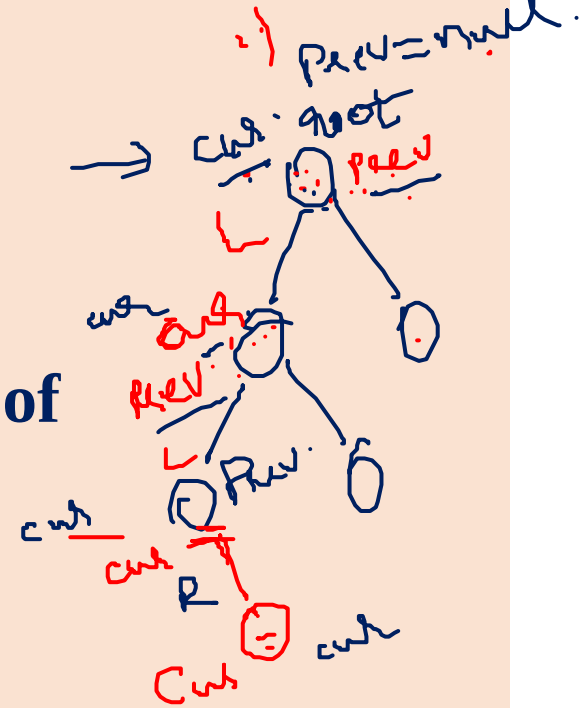
- To implement this, we make use of 2 pointer variables prev and cur. At any point prev points to the parent node and cur points to the child.

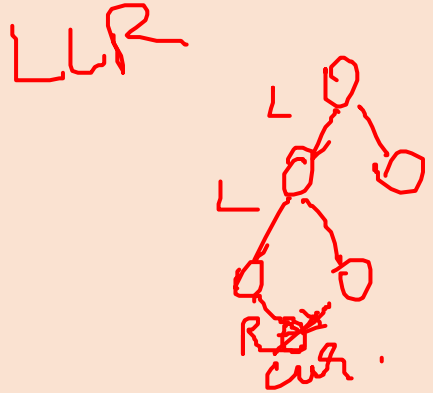
```
/*function to insert item into tree*/
NODEPTR insert(int item, NODEPTR root)
{
   NODEPTR temp, cur, prev;
   char direction[10];
   int i;
        temp=getnode();
        temp→info=item;
        temp→llink=temp→rlink=NULL;
    if(root==NULL)
          return temp;
   cout<<“enter the direction of insertion”;
   cin>>direction;
```



```
prev=NULL;
cur=root;
for(i=0;i<strlen(direction)&&cur!=NULL; i++)
{
   prev=cur;                    /*keep track of parent*/
   if(direction[i]==‘L’)        /*if direction is L, move left of
         cur=cur→llink;          current node*/
   else                         /*if direction is R, move right of
         cur=cur→rlink;          current node*/
}
If(cur!=NULL || i!=strlen(direction))
{
   cout<<“insertion not possible”;
   freenode(temp);
   return root;
}
```



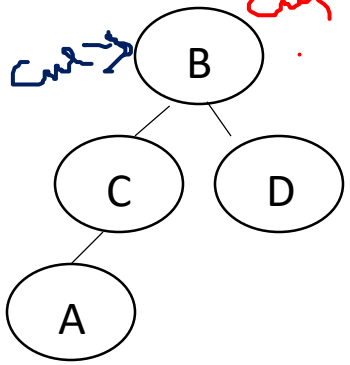




```
if (direction[i-1]==‘L’)          /* if last direction is ‘L’,
        prev→llink=temp;           point the llink of parent to new node*/
else                               /* else point rlink of parent to new  node*/
       prev→rlink=temp;
return root;
}
```

- Control comes out of for loop, **if i= strlen(direction)** or when **cur==NULL**.
- For insertion to be possible, control should come out when **cur==NULL** and **i = strlen(direction)** both at the same time.

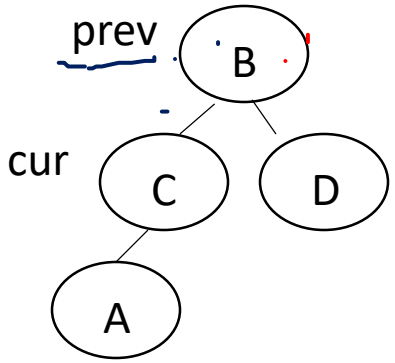
  i.e when we get the position to **insert(cur==NULL),** the string should be completed.



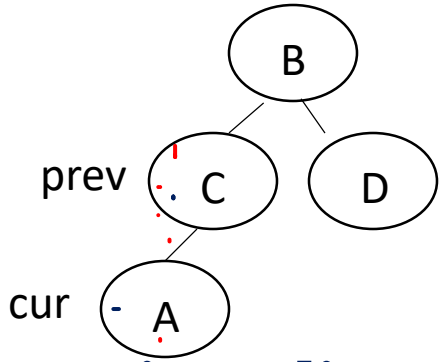
Let the direction string be "**LLR**". String length of string **is 3**.

Initially cur ==root and prev==NULL

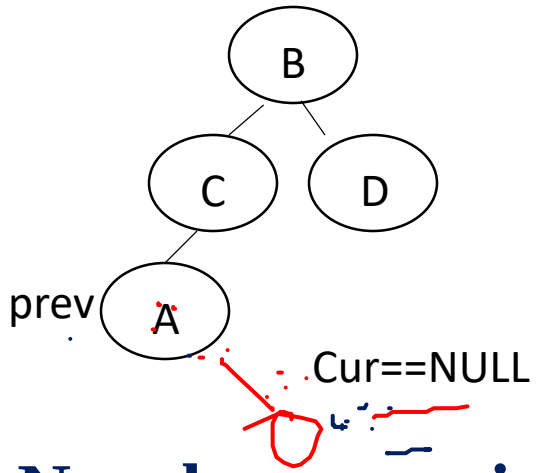
» Loop 1: **i=0 ,direction[0]='L'**



» Loop 2: **i=1, direction[1]='L'**



» Loop 3: **i=2, directiom[2]='R'**



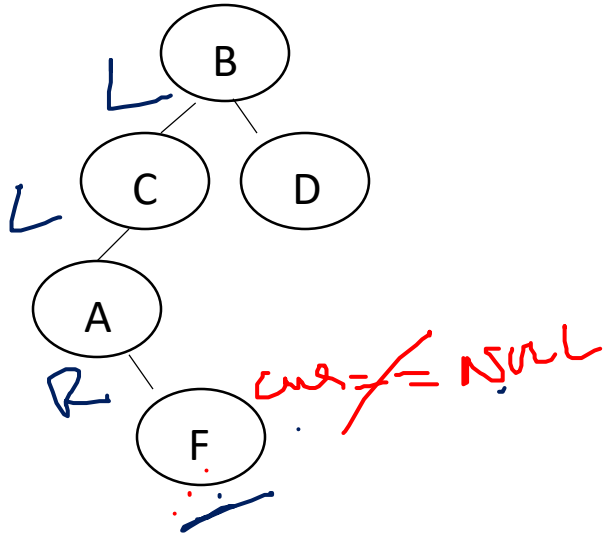
»

» **i=3, Now loop terminates since cur==NULL. Here i==3(i.e strlen(direction)) and cur==NULL. Hence insertion is possible**

» Now **direction[i-1]** is **direction[3-1]** which gives **'R'**.

» Hence **prev→rlink** is made to point to **new node**.

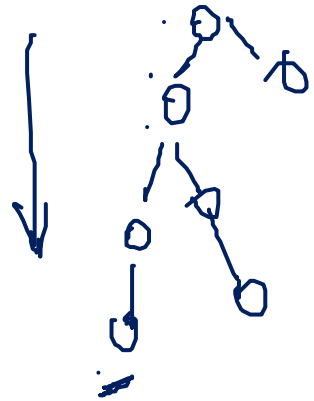
After insertion tree looks like

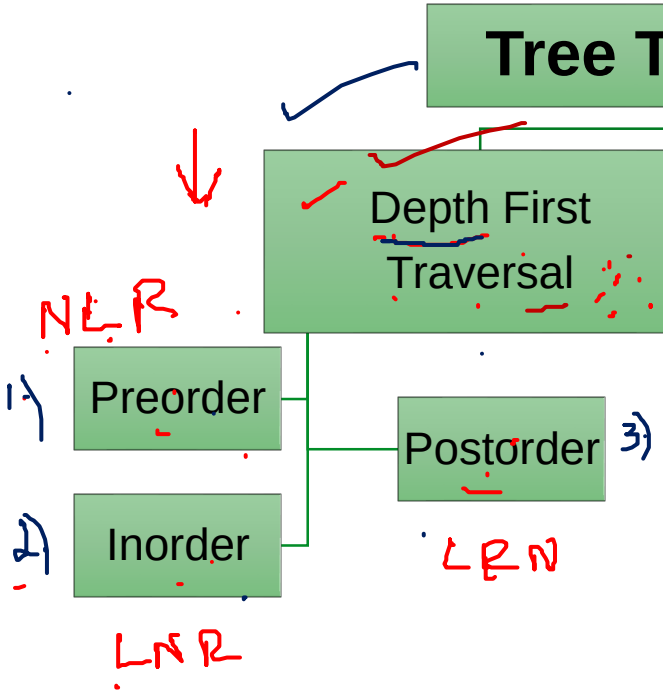


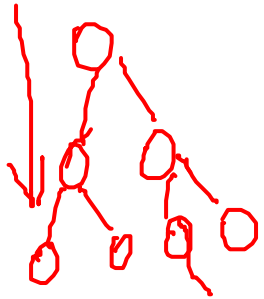
» If direction of insertion is "LLR" for the above tree. Here for loop terminates when i==strlen(direction) i.e 3. But at this point cur is not equal to NULL. Hence insertion is not possible.

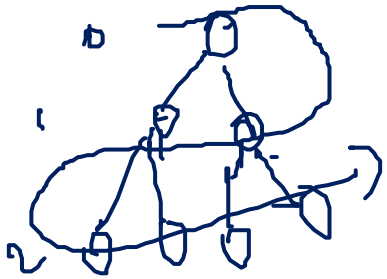
Binary Tree Traversal.

» A **binary tree traversal** requires that **each node of the tree be processed once and only once** in a predetermined sequence.









DEPTH FIRST TRAVERSAL ALGORITHM

- **USE Stack.**

Steps:

- Add root to the Stack.
- Pop out an element from Stack , process it, and add its right and left children to stack.
- Repeat the above two steps until the Stack id empty.

DEPTH FIRST SEARCH ALGORITHM

Put the root node onto the stack

while (stack is not empty)

do

remove a node from the stack;

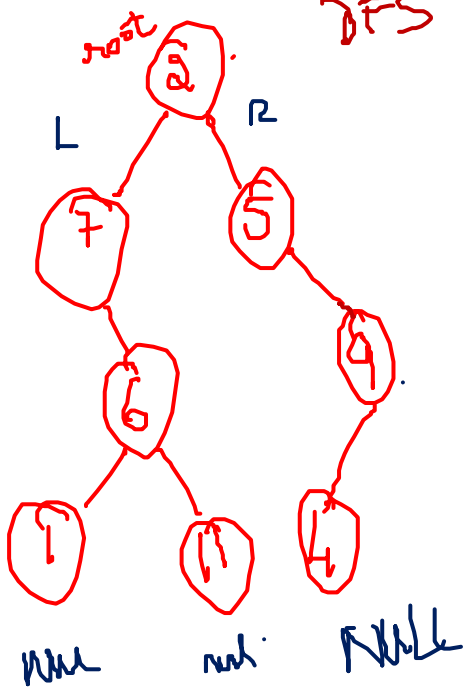
if (node is a goal node)

return success;

Put its **right and left children** onto the stack;

done

return failure;



DFS

Stack.

top

1) 2

2) 5 7

3) 5 6

4) 5 11 1

5) 5 11

6) 5

7) 9

8) 4

9)

o/p string.

2 2==9 X

2,7 7==9 X

2,7,6 6==9 X

2,7,6,1 1==9 X

2,7,6,1,11 11==9 X

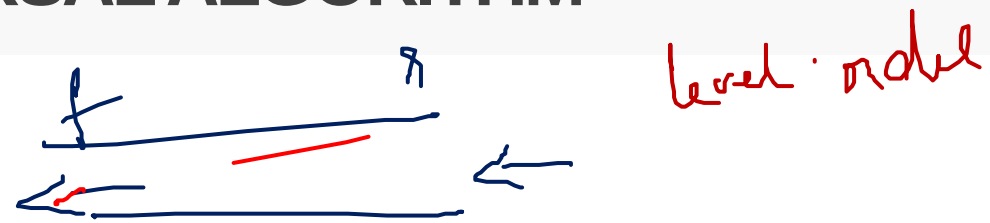
2,7,6,1,11,5 5==9 X

2,7,6,1,11,5,9 9==9 ✓

2,7,6,1,11,5,9,4

Goal node := 9 ==

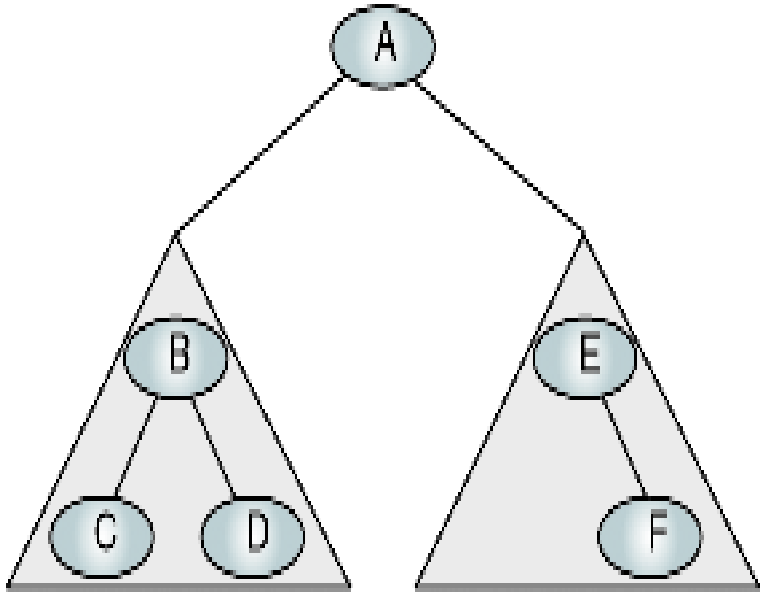
BREADTH FIRST TRAVERSAL ALGORITHM

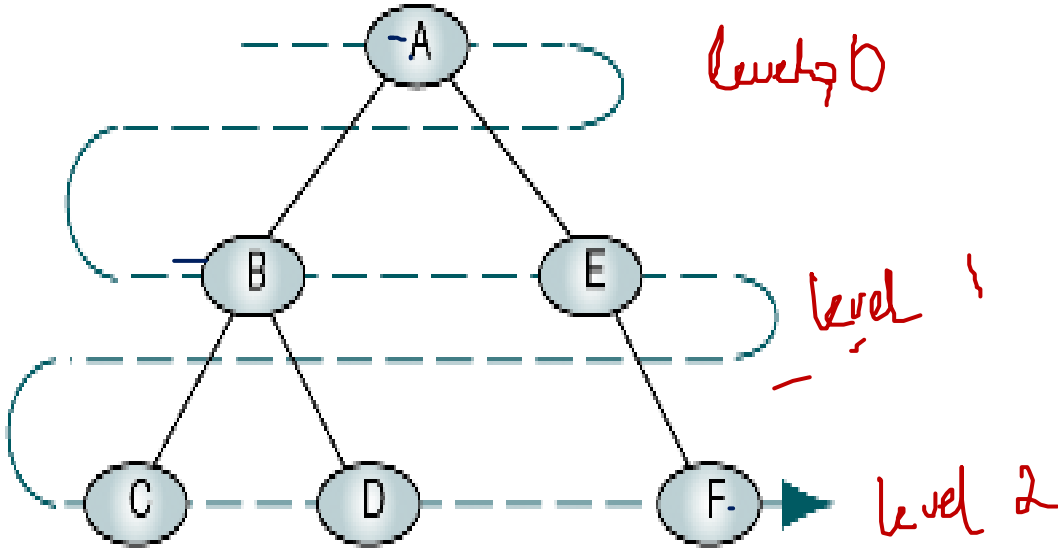


- **USE Queue.**

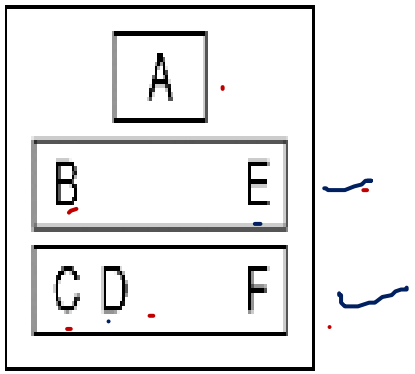
Steps:

- Add root to the **Queue.**
- Pop out an element from queue, process it, and add its left and right children to the queue.
- Repeat the above two steps until the queue is empty.





(b) "Walking" order

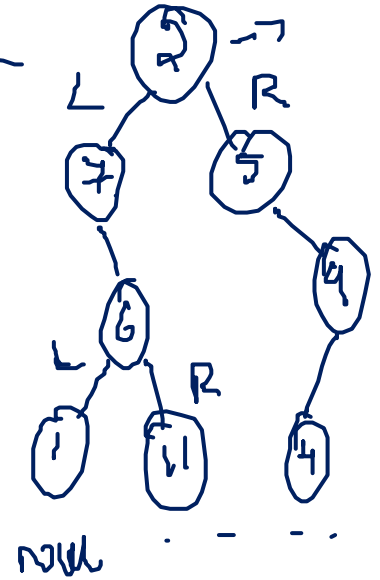


(a) Processing order

FIGURE 6-14 Breadth-first Traversal

BREADTH FIRST SEARCH ALGORITHM.

```
Add root to queue
while (queue is not empty)
do
        remove a node from the queue;
        if (node is a goal node)  return success;
        put its left and right children  onto the queue;
done
return failure;
```



Queue.

1) 2

2) 7 | 5

3) 5 | 6

4) 6 | 9

5) 9 | 1 | 11

6) 1 | 11 | 4

7) 11 | 4

8) 4

9)

O/P

2

2,7

2,7,5

2,7,5,6

2,7,5,6,9

2,7,5,6,9,1

2,7,5,6,9,1,11,4

2,7,5,6,9,1,11,4.

2 == 9 ✗

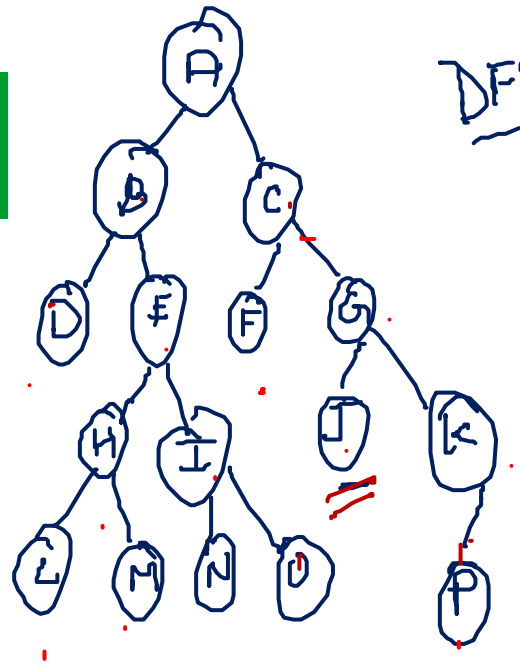
7 == 9 ✗

5 == 9 ✗

6 == 9 ✗

9 == 9 ✓

Goal node = 9.



DFS

BFS

DFS → STACK

BFS → queue

Goal node = I

Key element

I

Depth First Traversal.

- In the **depth-first traversal** processing proceeds along a path from the **root through one child to the most distant descendant** of that first child before processing a second child.

OR

- We process all the descendants of a child before going on to the next child.

- Is visiting each node exactly once systematically one after the another.

There are 3 main traversals techniques

- **Inorder traversal**
- **Preorder traversal**
- **Postorder traversal**

Preorder

NLR

Inorder

LNR

Postorder

LRN

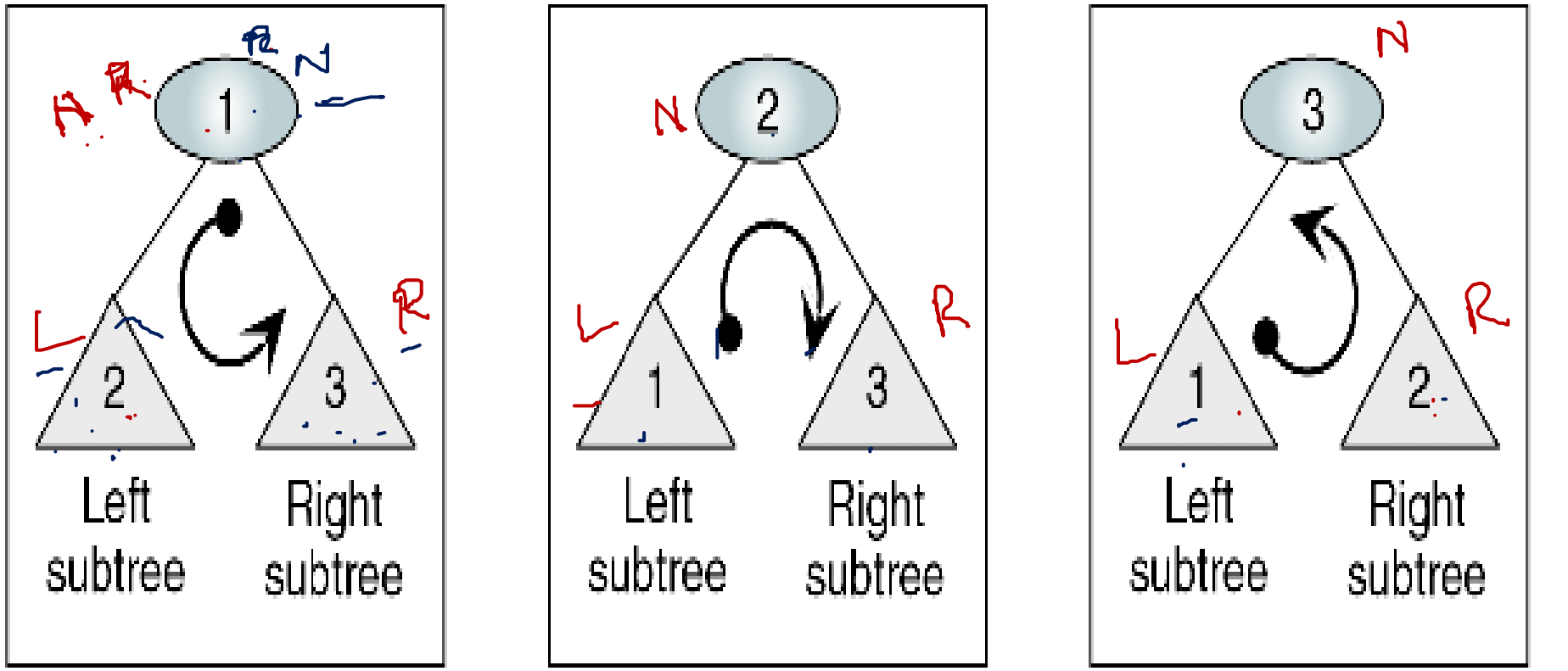


FIGURE 6-8 Binary Tree Traversals

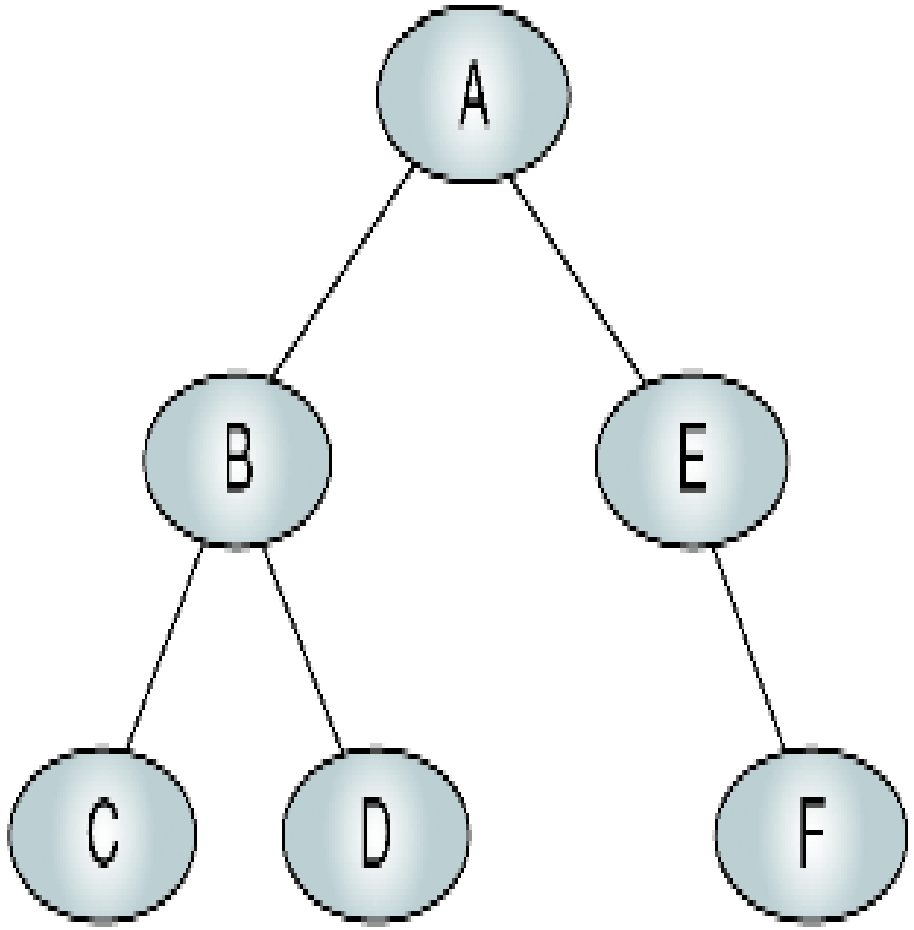
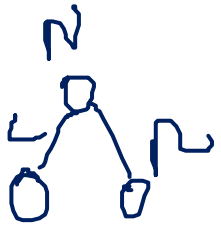


FIGURE 6-9 Binary Tree for Traversals

Preorder Traversal



NLR

1. Process the node.
2. Traverse the left subtree in preorder.
3. Traverse the right subtree in preorder.

- In preorder, we first visit the node, then move towards left and then to the right recursively.

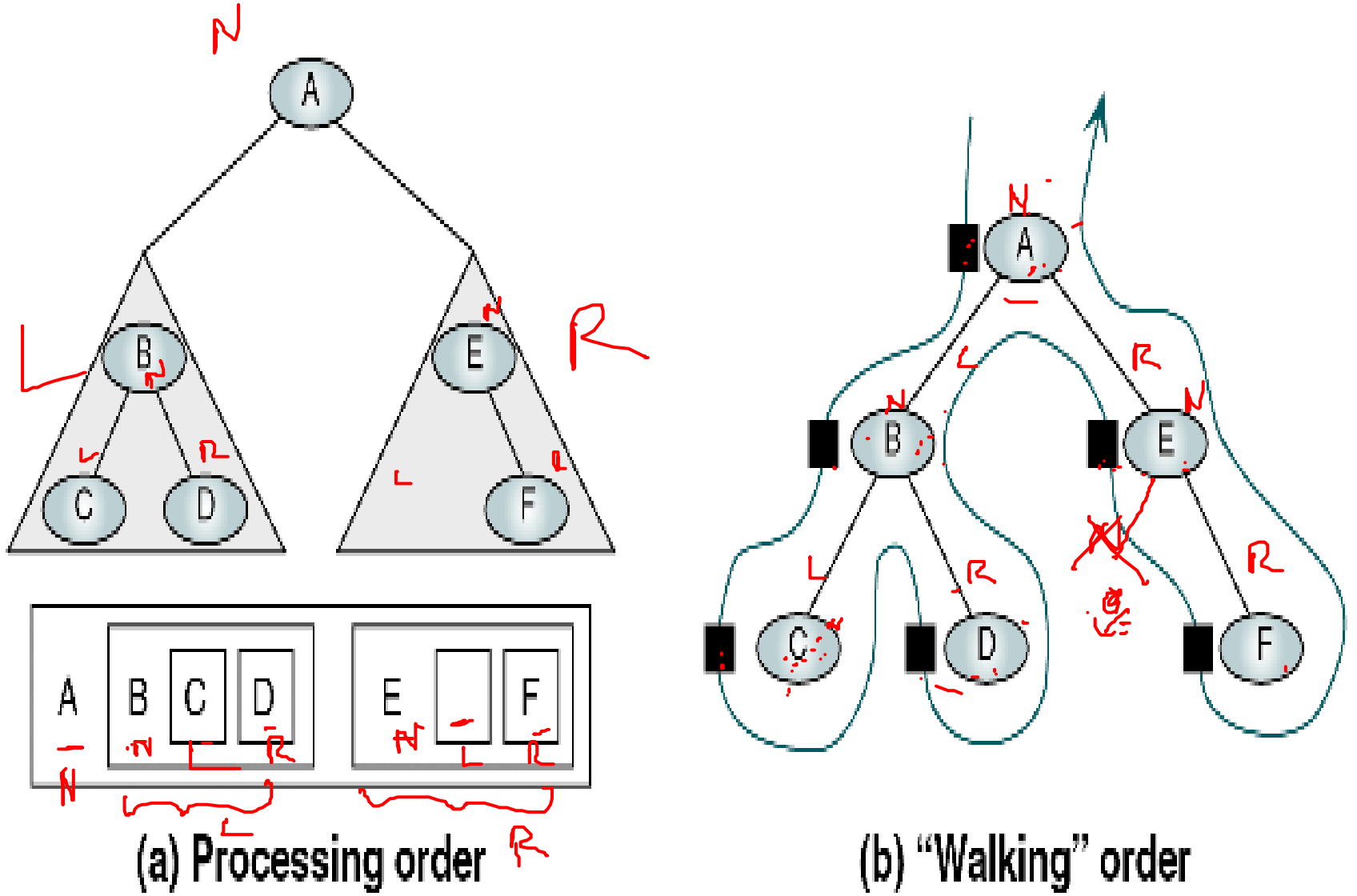


FIGURE 6-10 Preorder Traversal—A B C D E F

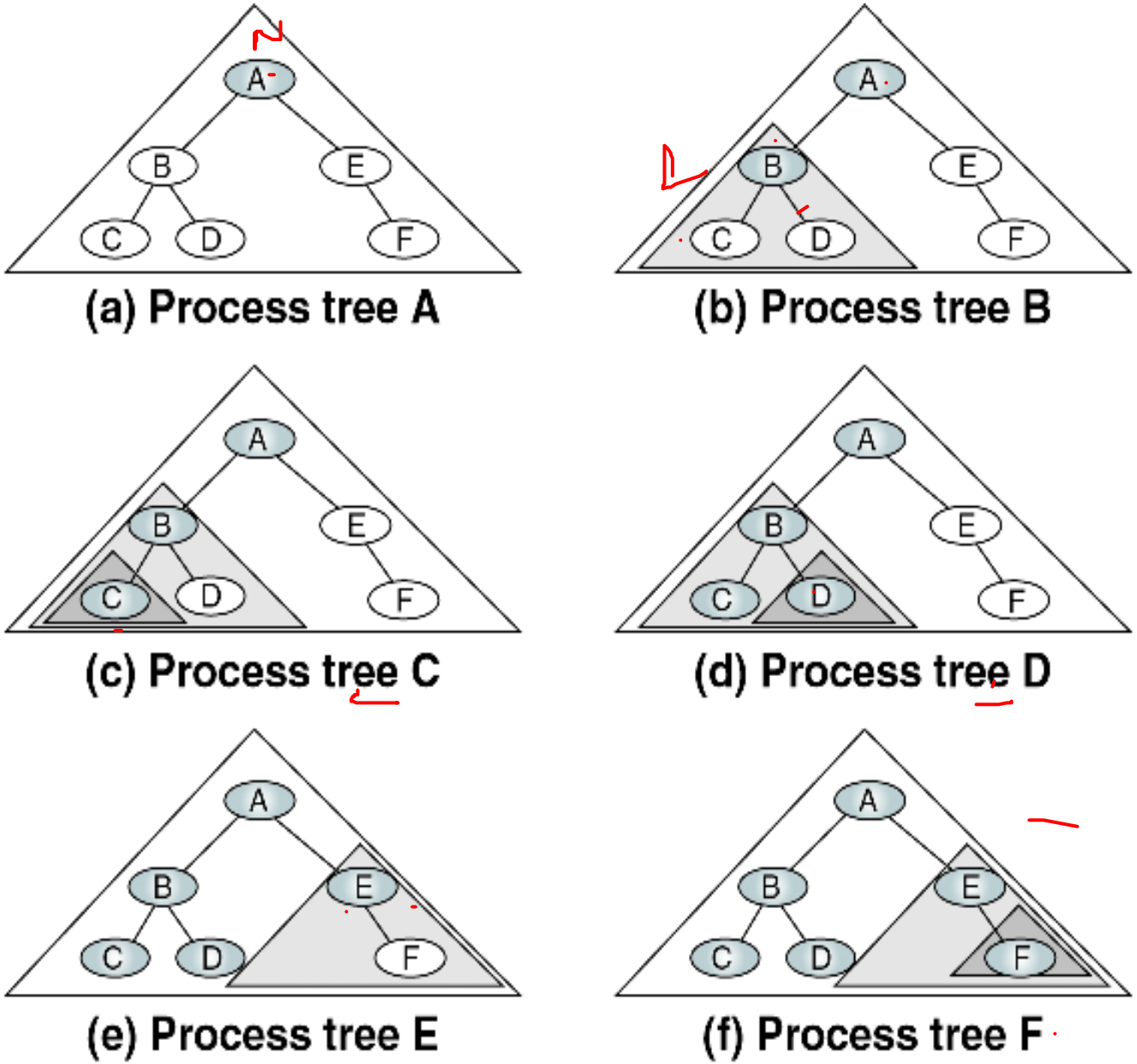


FIGURE 6-11 Algorithmic Traversal of Binary Tree

ALGORITHM 6-2 Preorder Traversal of a Binary Tree

```
Algorithm preOrder (root)
Traverse a binary tree in node-left-right sequence.
   Pre  root is the entry node of a tree or subtree
   Post each node has been processed in order
1  if (root is not null)
   1  process (root)
   2  preOrder (leftSubtree)
   3  preOrder (rightSubtree)
2  end if
end preOrder
```

Preorder Traversal (recursive version)

```
void preorder(node *root)
/* preorder tree traversal */
{
     if (root) {
        cout<<root->data;
        preorder(root->left_child);
        predorder(root->right_child);
     }
}
```

+**/ABCDE

Inorder traversal

1. Traverse the left subtree in inorder.
2. Process the root node.
3. Traverse the right subtree in inorder.

- In inorder traversal, move towards the left of the tree(till the leaf node), display that node and then move towards right and repeat the process

- Since same process is repeated at every stage, recursion will serve the purpose.

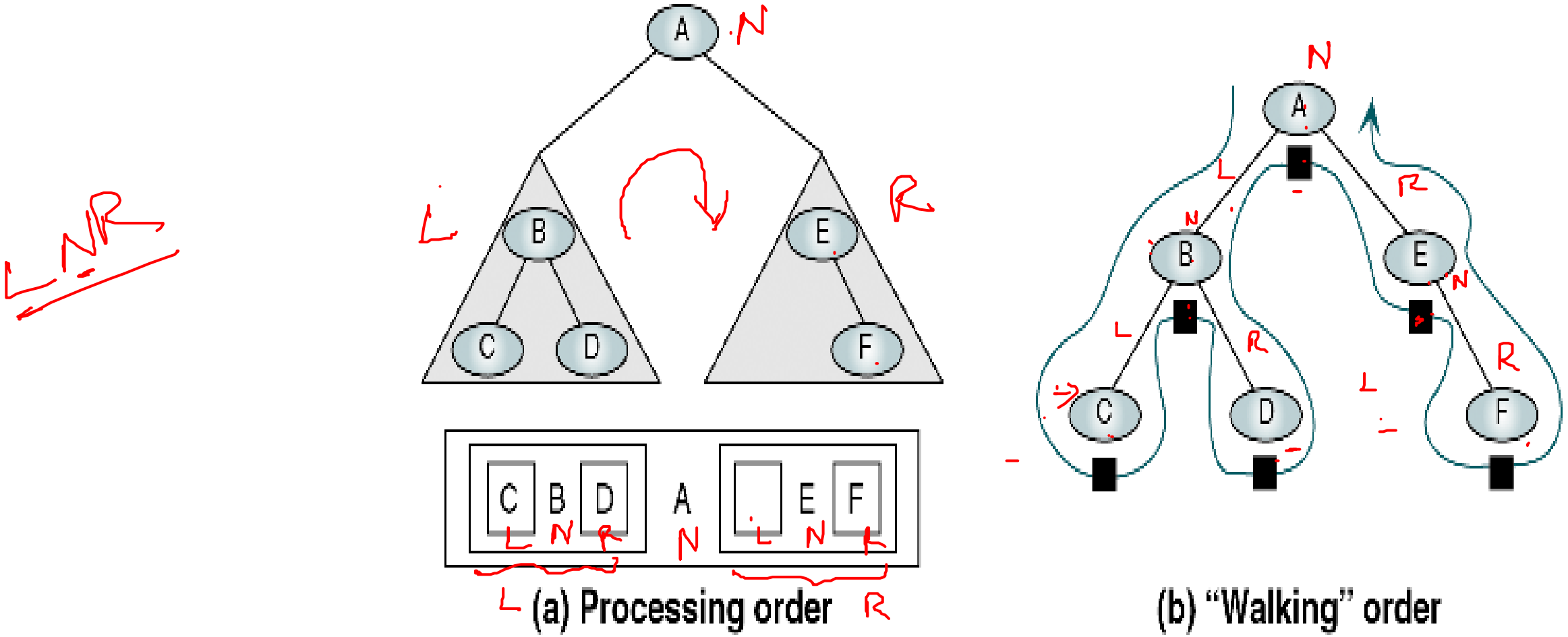


FIGURE 6-12 Inorder Traversal—C B D A E F

ALGORITHM 6-3 Inorder Traversal of a Binary Tree

```
Algorithm inOrder (root)
Traverse a binary tree in left-node-right sequence.
   Pre  root is the entry node of a tree or subtree
   Post each node has been processed in order
1  if (root is not null)
   1  inOrder (leftSubTree)
   2  process (root)
   3  inOrder (rightSubTree)
2  end if
end inOrder
```

Inorder Traversal (recursive version)

```
void inorder(node *root)
/* inorder tree traversal */
{
     if (ptr) {
        inorder(root->left_child);
        cout<<root->data;
        indorder(root->right_child);
     }
}
```

A / B * C * D + E

Post order traversal

1. Traverse the left subtree in postorder. L
2. Traverse the right subtree in postorder. R
3. Process the root node. N

- In post order traversal, we first traverse towards left, then move to right and then visit the root. This process is repeated recursively.

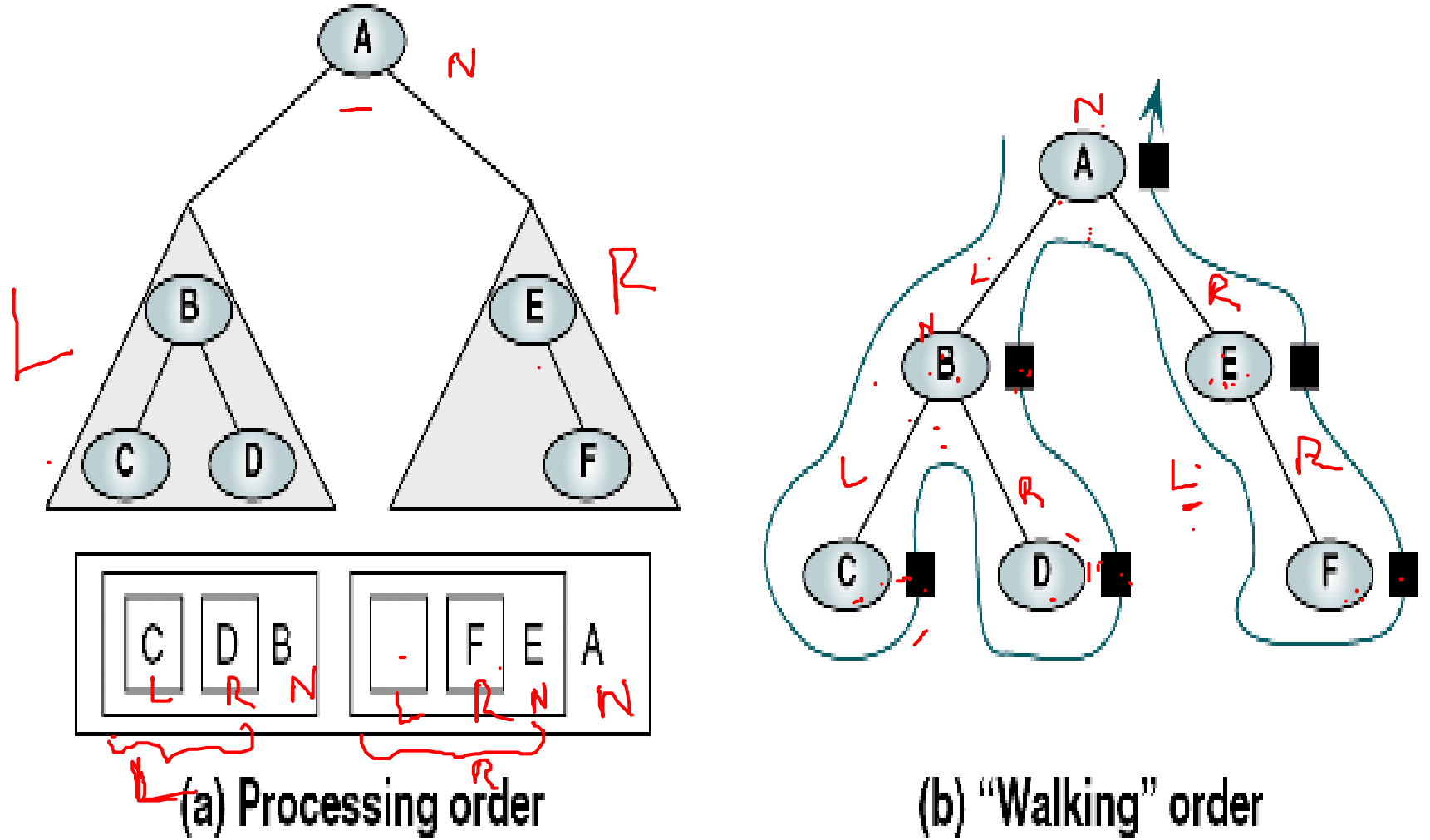


FIGURE 6-13 Postorder Traversal—C D B F E A

ALGORITHM 6-4 Postorder Traversal of a Binary Tree

```
Algorithm postOrder (root)
Traverse a binary tree in left-right-node sequence.
   Pre  root is the entry node of a tree or subtree
   Post each node has been processed in order
1  if (root is not null)
   1 Lpostorder (left subtree)
   2 Rpostorder (right subtree)
   3 Nprocess (root)
2  end if
end postOrder
```

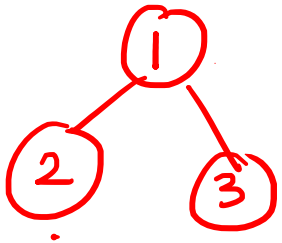
Postorder Traversal (recursive version)

```
void postorder(node *root)
/* postorder tree traversal */
{
     if (root) {
        postorder(root->left_child);
        postdorder(root->right_child);
        cout<<root->data;
     }
}
```

A B / C * D * E +

Searching:

- Searching an item in the tree can be done while traversing the tree in inorder, preorder or postorder traversals.
- While visiting each node during traversal, instead of printing the node info, it is checked with the item to be searched.
- If item is found, search is successful.

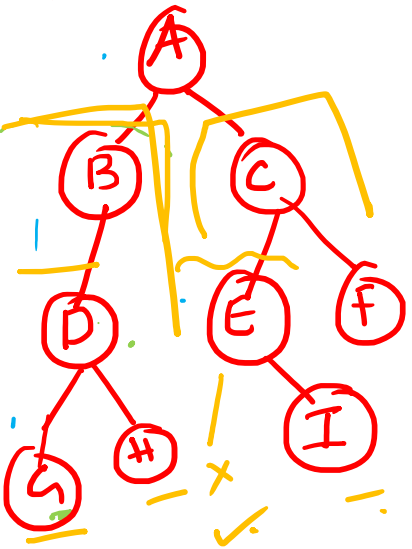


root

Preorder (PLR) ⇒ 1 2 3

Inorder (LPR) ⇒ 2 1 3

Postorder (LRP) ⇒ 2 3 1



Preorder (PLR)

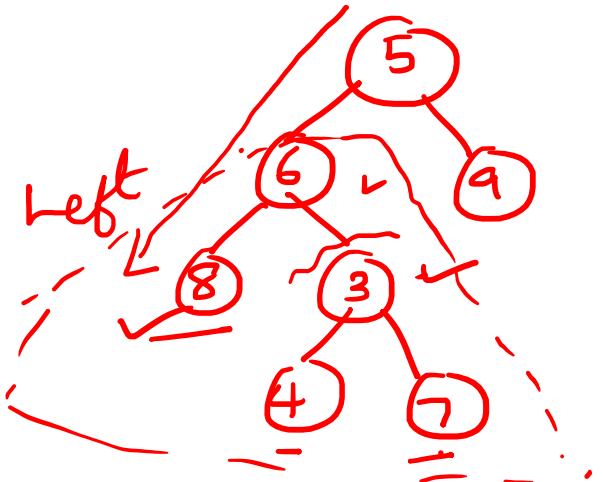
→ A B D G H C E I F

Inorder (LPR)

G D H B A E I C F

Postorder (LRP)

G H D B I E F C A



Preorder (PLR)

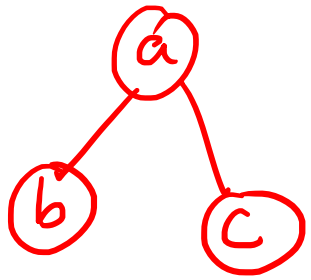
5 6 8 3 4 7 9

Inorder (LPR)

8 6 4 3 7 5 9

Postorder (LRP)

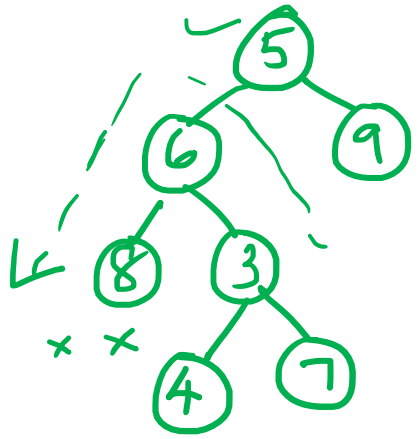
8 4 7 3 6 9 5



Preorder (PLR) ⇒ a b c

Inorder (LPR) ⇒ b a c

Postorder (LRP) ⇒ b c a



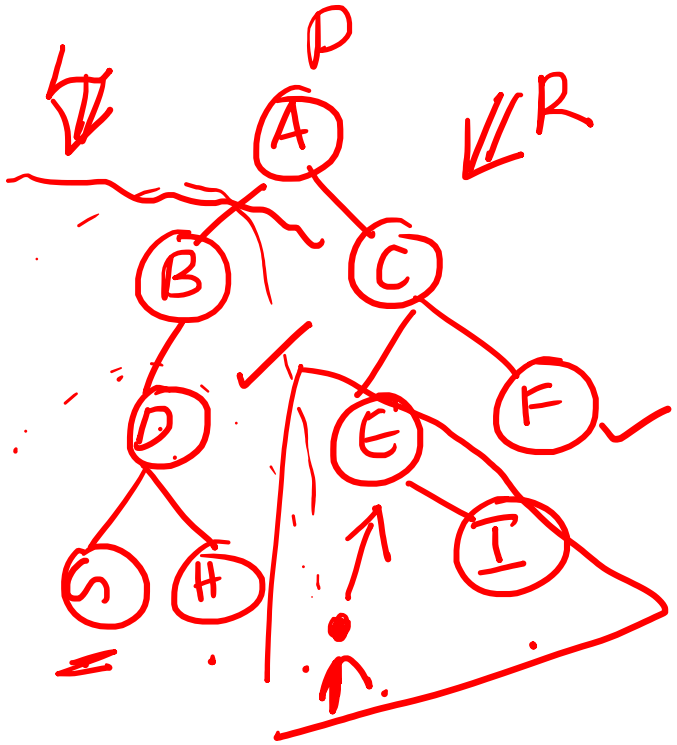
PRE 5 6 8 3 4 7 9

IN LPR

8 6 4 3 7 5 9

POST — LRP

8 4 7 3 6 9 5



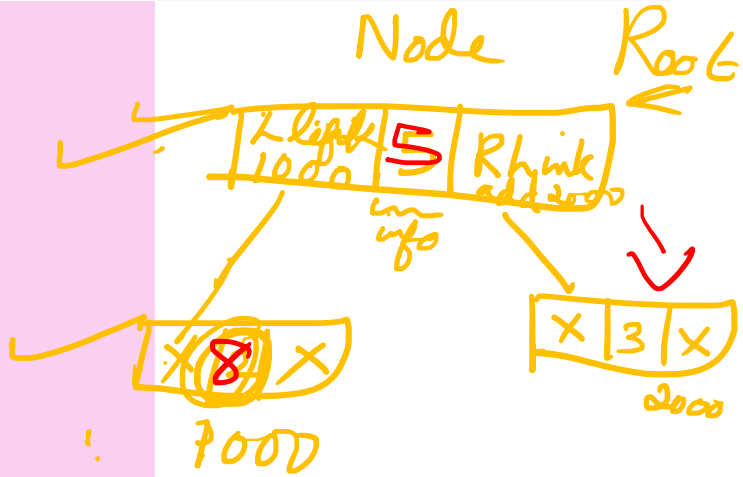
Preorder (PLR) ⇒ A B D G H C E I F

Inorder (LPR) ⇒ G D H B A E I C F

Postorder (LRP) ⇒ G H D B I E F C A

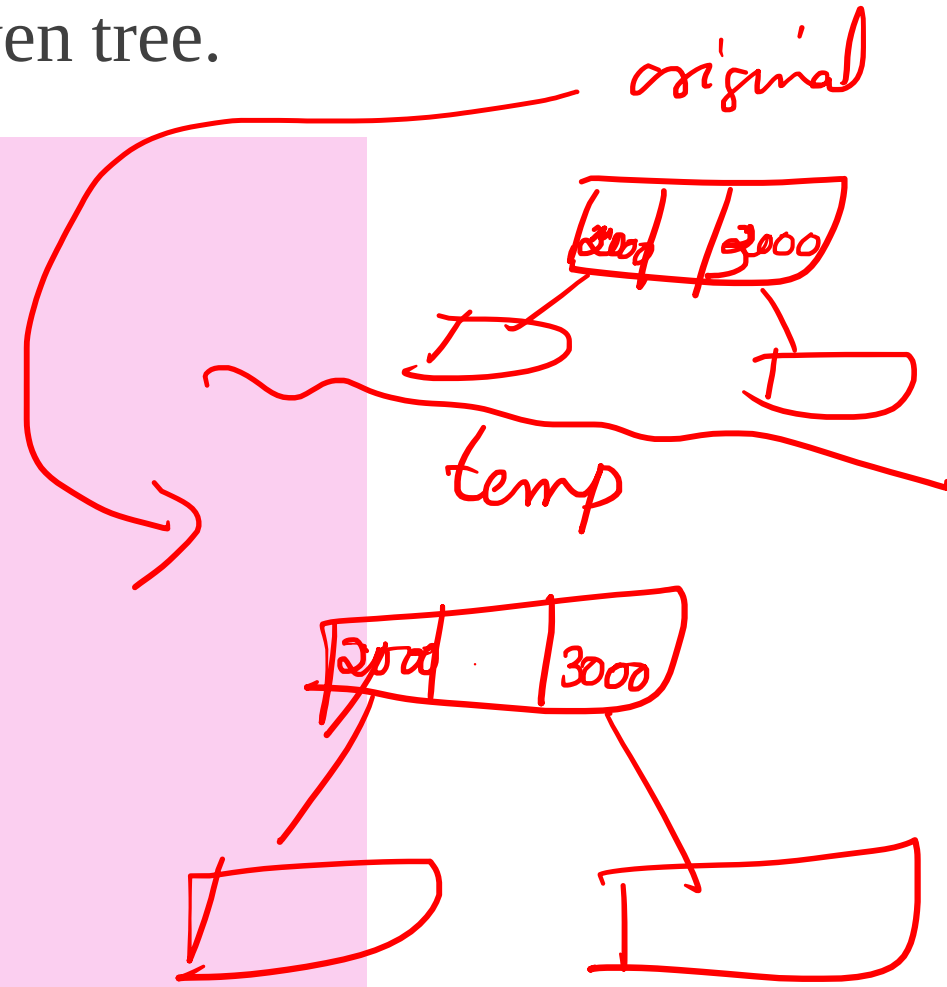
Searching

```
void search(int item, NODEPTR root, int *flag)
{  if(root!=NULL)
   {       if(item==root→info) {
                   *flag=1
                   return;
           }
      search(item, root→llink, flag);
      if (!(*flag)) search(item, root→rlink, flag);
   }}
```



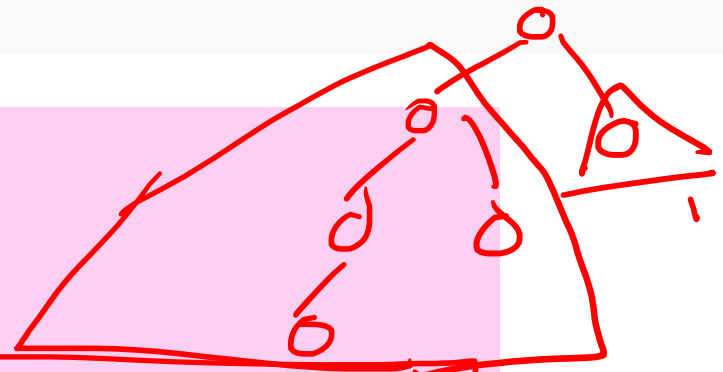
Copying a tree: Getting the exact copy of the given tree.

```
/*recursive function to copy a tree*/
NODEPTR copy (NODEPTR root)
{
        NODEPTR temp;
        if(root == NULL)
                return NULL;

        temp=getnode();

        temp→info=root→info;
        temp→llink=copy(root→llink);
        temp→rlink=copy(root→rlink);
        return temp;
}
```



Recursive function to find the height of a tree

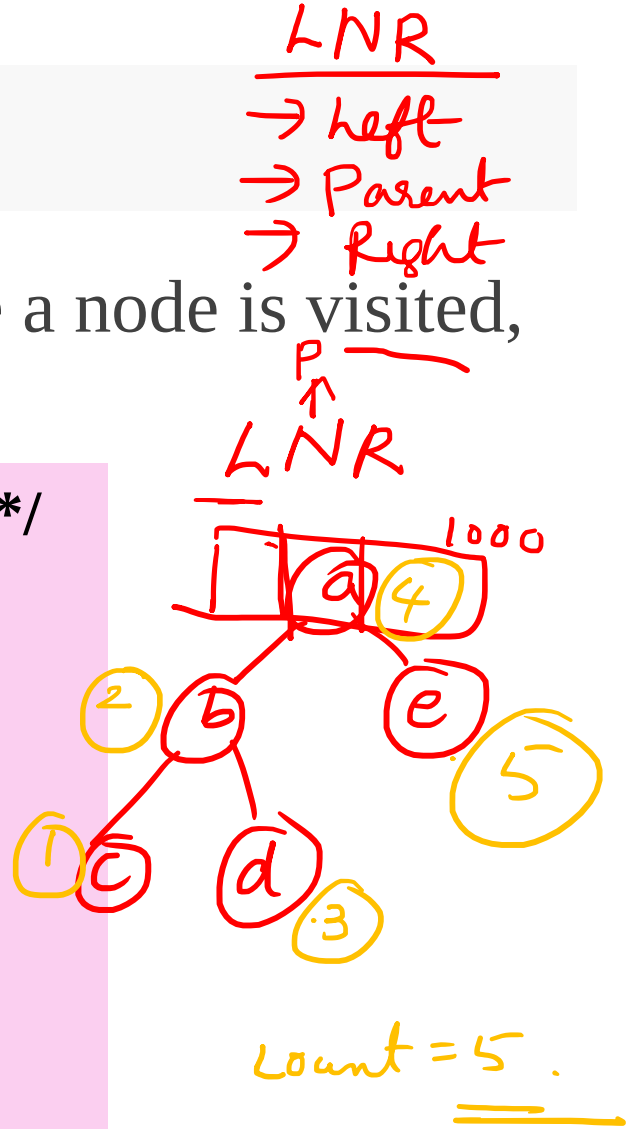
```
Int height (NODEPTR root)
{
        if(root==NULL)
                return 0;
        return( 1+ max(height (root→llink), height(root→rlink)));
}

/*max function*/
Int max(int a, int b)
{
        if(a>b)
                return a;
        else
                return b;
}
```



Counting the number of nodes in a tree:

- Traverse the tree in any of the 3 techniques and every time a node is visited, count is incremented.

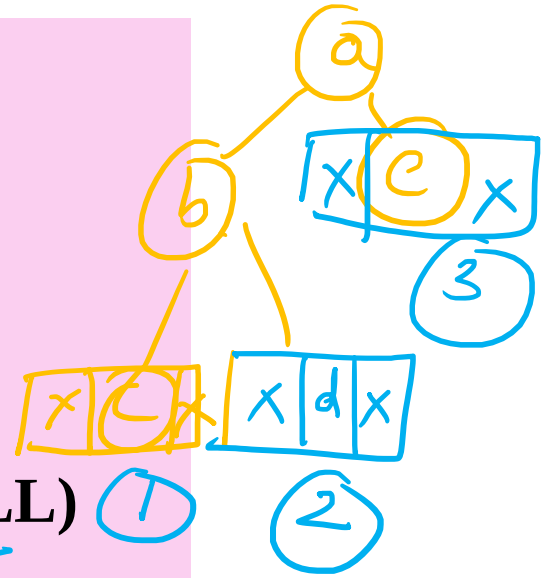
```
/*counting number of nodes using inorder technique*/
Void count_nodes( NODEPTR root)
{
        if(root!=NULL)
        {
            count_nodes(root→llink);
              count++;
            count_nodes(root→rlink);
        }
}
```



Counting the number of leaf nodes in a tree:

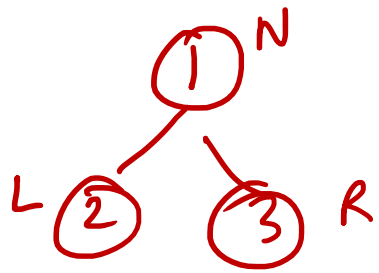
- Traverse the tree in any of the 3 techniques and every time a node is visited, check whether the right and left link of that node is NULL. If yes, count is incremented.

```
/*counting number of leaf nodes using inorder technique*/
Void count_leafnodes( NODEPTR root)
{
        if(root!=NULL)
        {
            count_leafnodes(root→llink);
            if(root→llink==NULL && root→rlink==NULL)
                count++;
            count_leafnodes(root→rlink);
        }
}
```



More Examples.

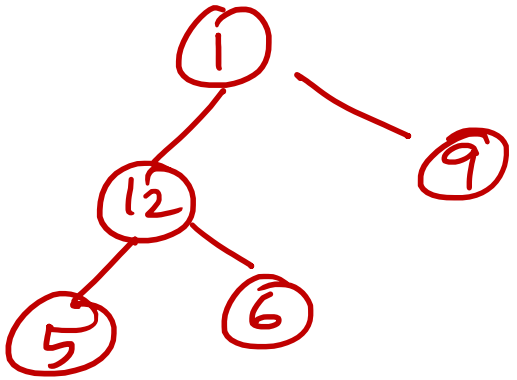
RLR LNR LRN



I 1, 2, 3

II 2, 1, 3

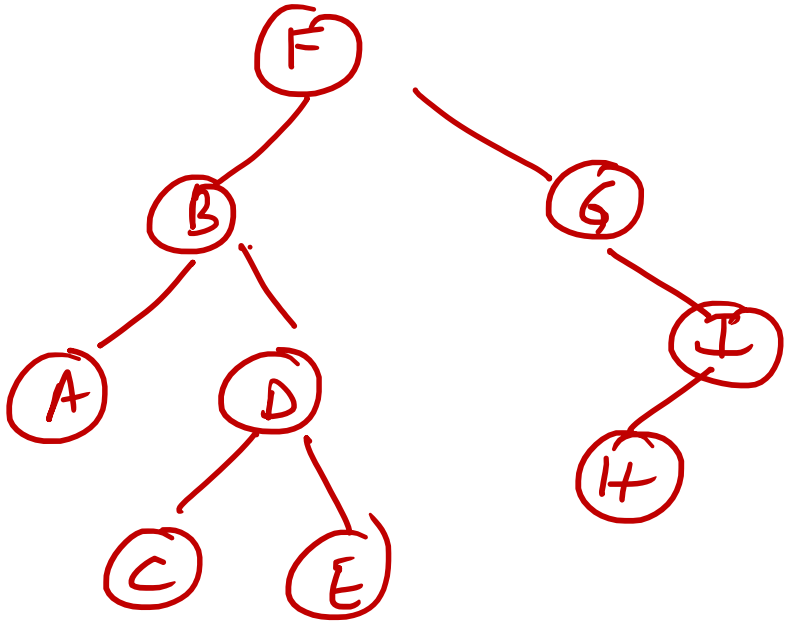
III 2, 3, 1



I 1, 12, 5, 6, 9

II 5, 12, 6, 1, 9

III 5, 6, 12, 9, 1

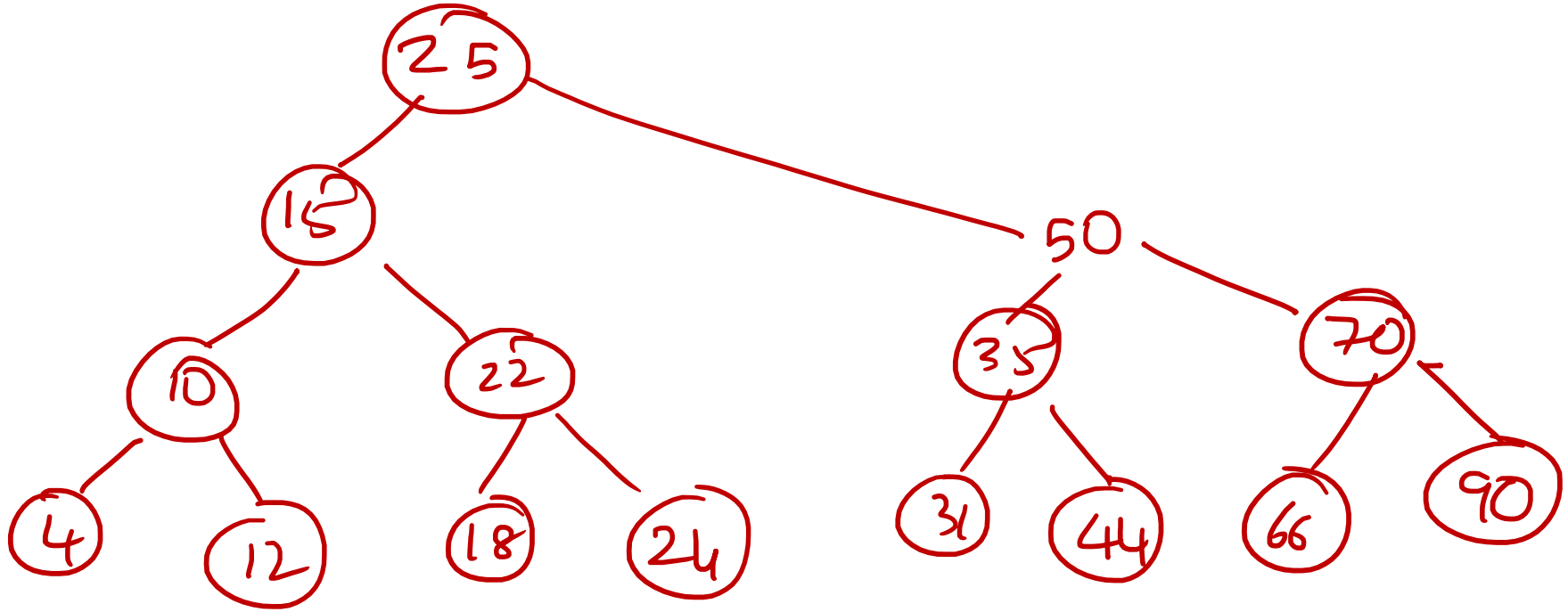


I F, B, A, D, C, E, G, I H

II A, B, C, D, E, F, G, H, I

III A, C, E, D, B, H, I, G, F

More Examples.

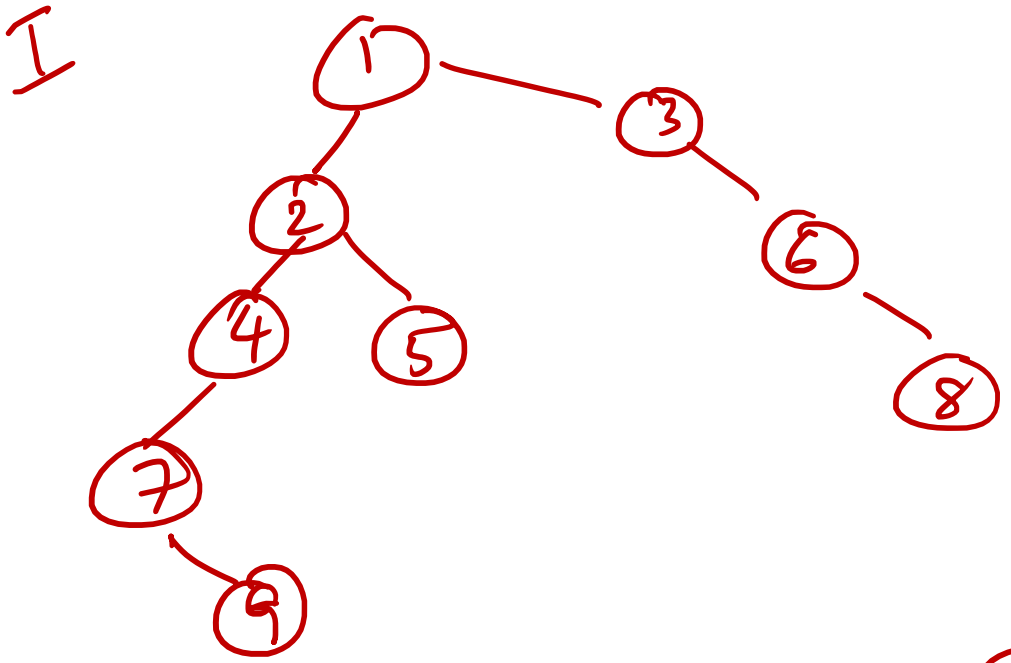


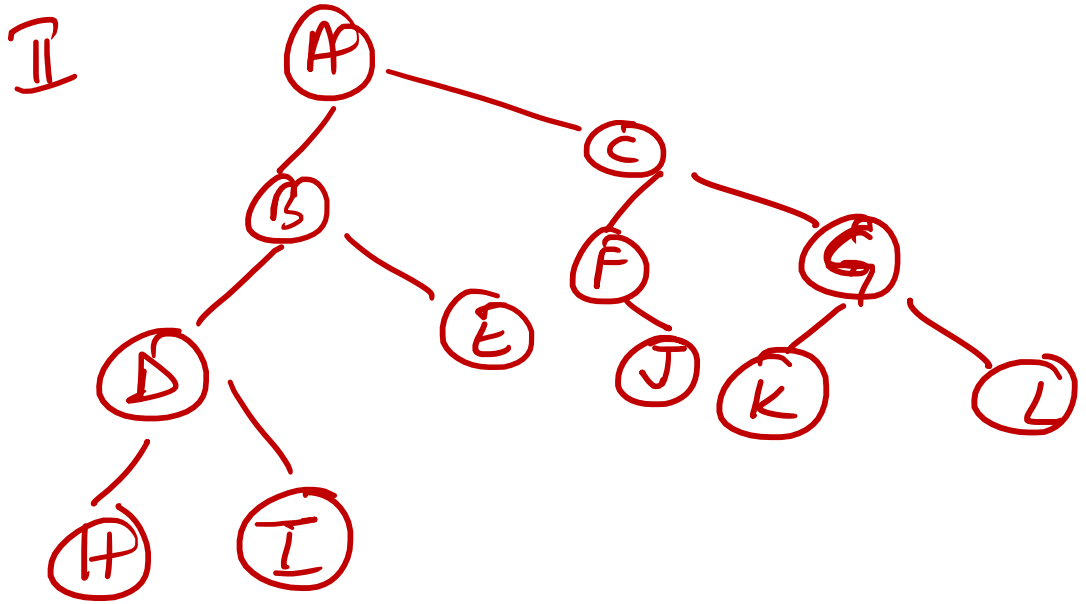
I 25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90.

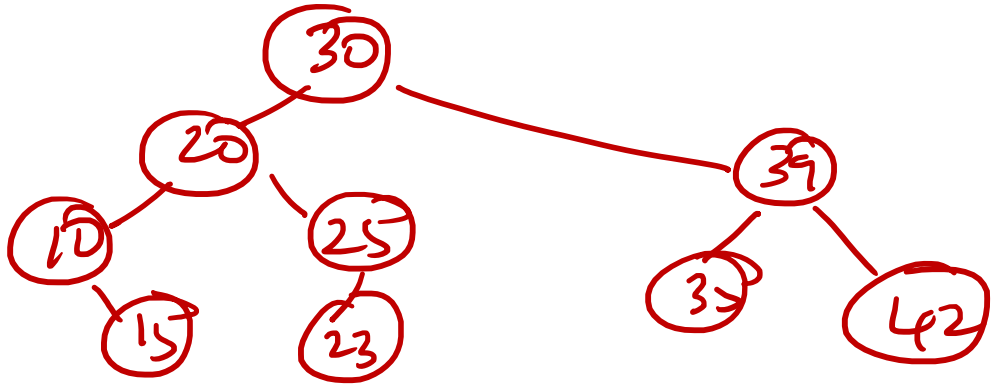
II 4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

III 4, 12, 10, 18, 12, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

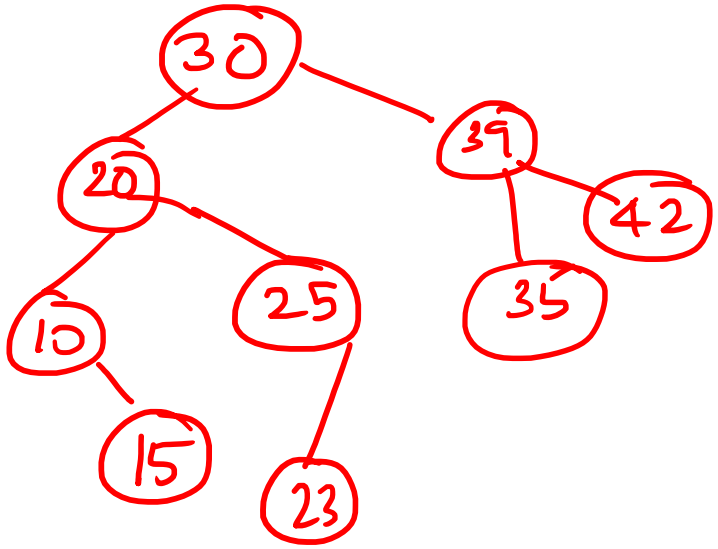
More Examples.







NLR.

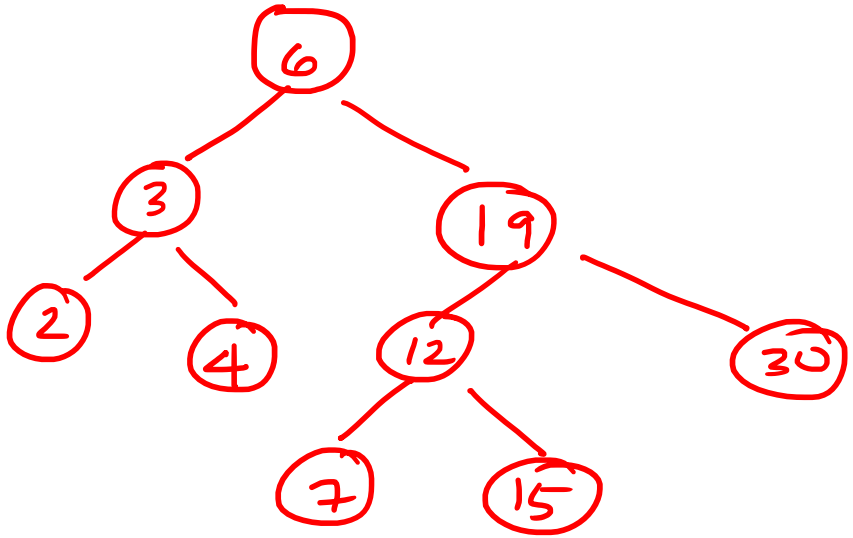


I) 30, 20, 10, 15, 25, 23, 39, 35, 42

II) 10, 15, 20, 23, 25, 30, 35, 39, 42

III) 15, 10, 23, 25, 35, 42, 39, 30, 20

LRN

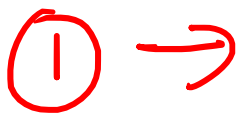


I 6, 3, 2, 4, 19, 12, 7, 15, 30

II 2, 3, 4, 6, 7, 12, 15, 19, 30

III 2, 4, 3, 7, 15, 12, 30, 19, 6.

BINARY TREE CONSTRUCTION.



Inorder and Preorder

Inorder and Postorder

Preorder and Postorder

Homework :- Inorder - E A C K F H D B G
Preorder - F A E K C D H G B

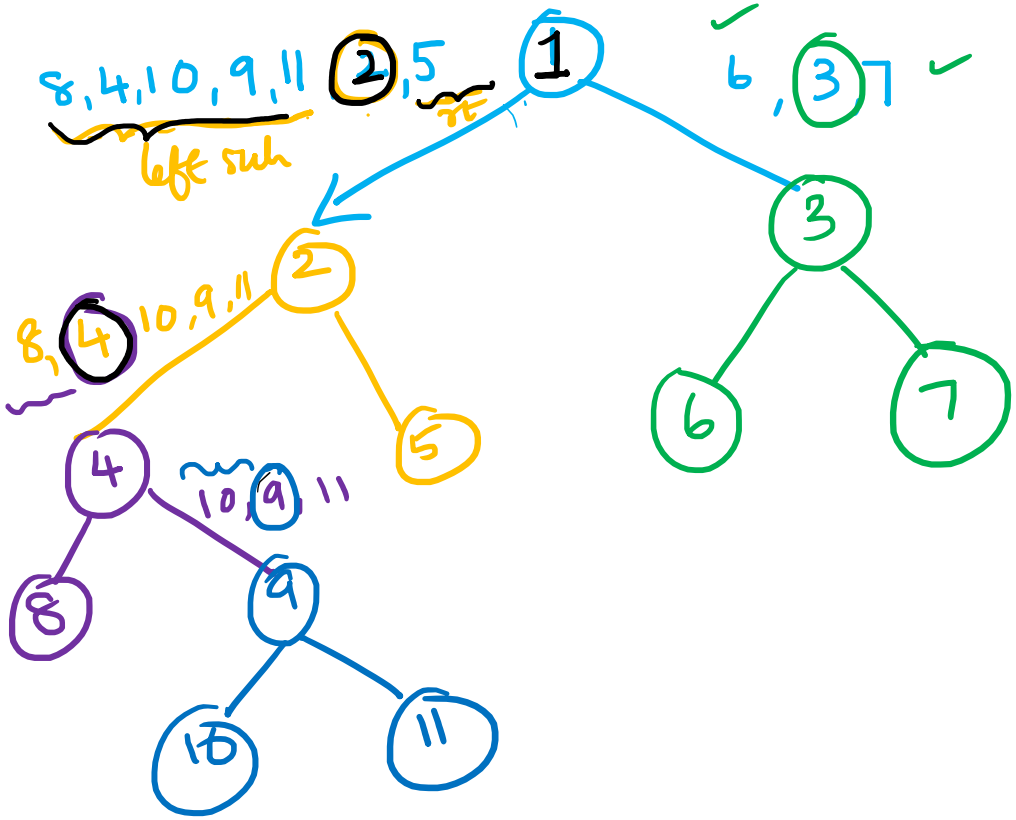
1. Binary Tree construction from Preorder and Inorder – Example 1.

(PLR) Preorder :- 1 2 4 8 9 10 11 5 3 6 7

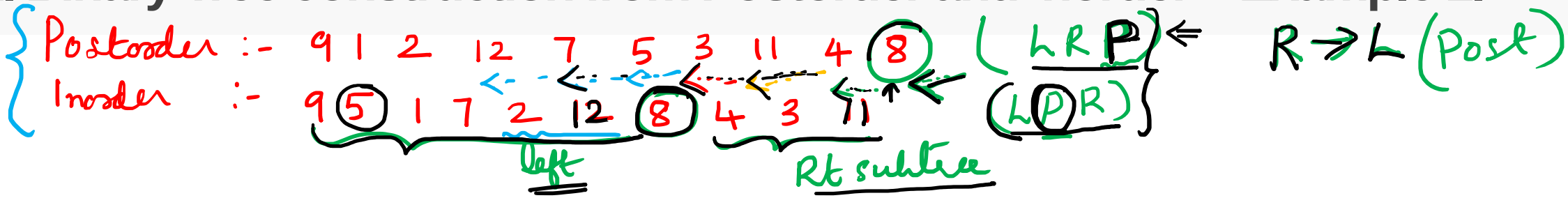
(LPR) Inorder :- 8 4 10 9 11 2 5 1 6 3 7

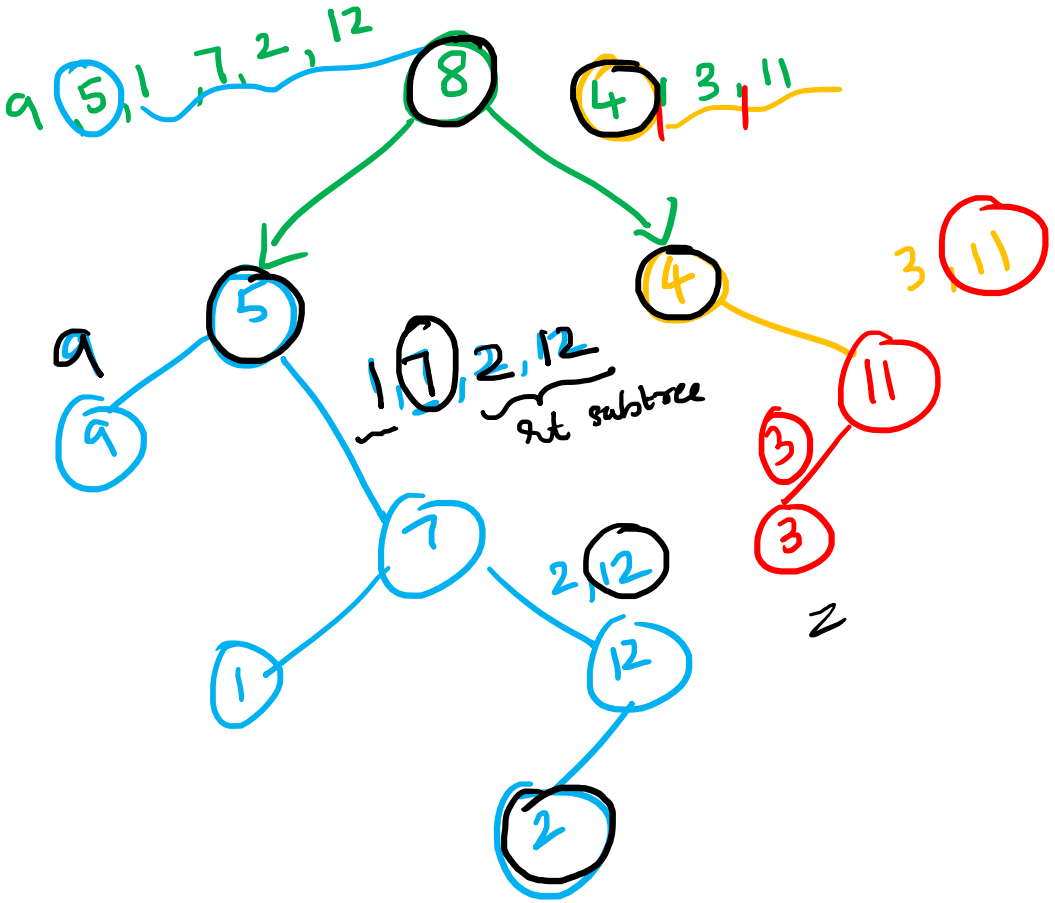
L → R (Preorder)

left subtree | rt subtree



1. Binary Tree construction from Postorder and Inorder – Example 1.





HW

Inorder
E A C K F H D B G

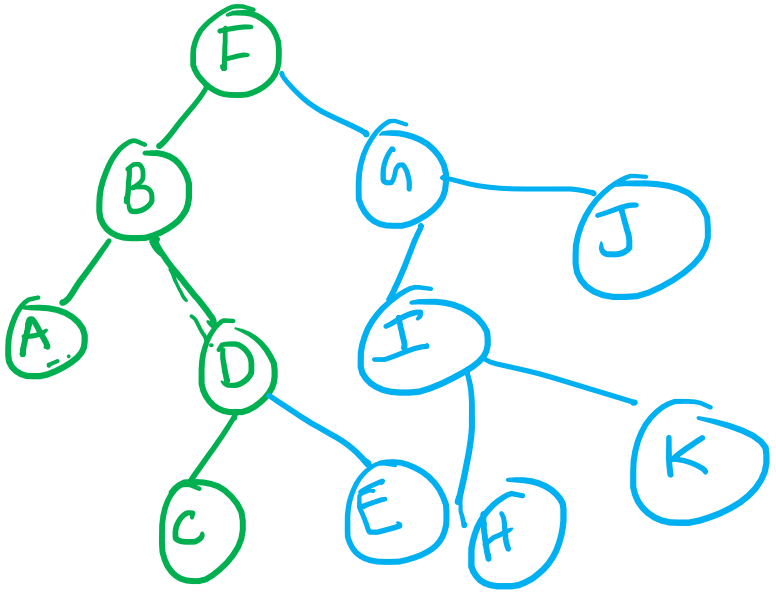
Postorder
E C K A H B G D F

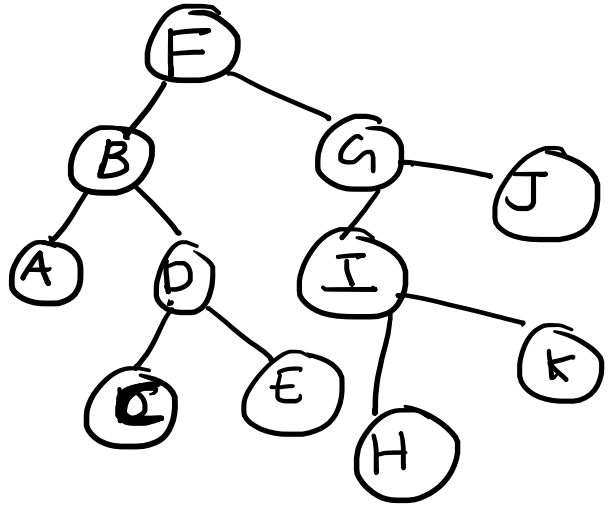
Full binary tree (2-tree)

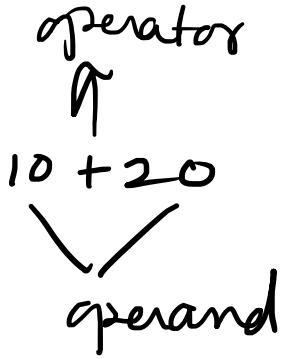
1. Binary Tree construction from Preorder and postorder – Example 1.

PLR)) Preorder :- F B A D C E G I H K J

LRP)) Postorder :- A C E D B H K I J G F





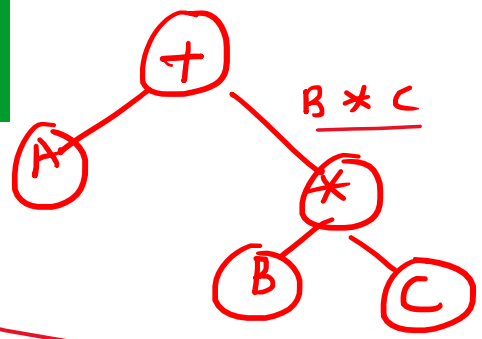


Expression Tree

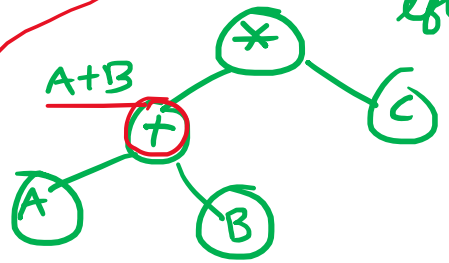
- Expression tree is a binary tree with the following properties:
  - Each leaf is an operand
  - The root and internal nodes are operators
  - Sub-trees are sub-expressions with the root being an operator

① INFIX :- A + B * C

-->



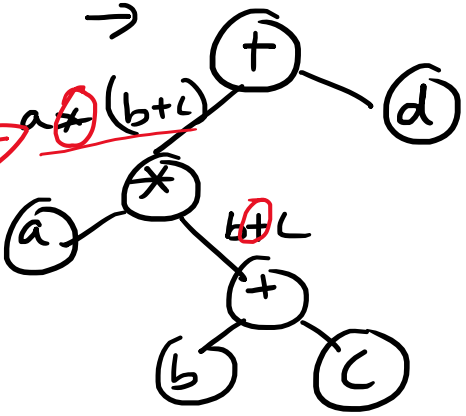
② (A+B) * C



BODMAS

		Associativity
1)	↑	- R-L
2)	* /	- L-R
3)	+ -	- L-R

③ a*(b+c)+d



④ HW

a+b*c -d /e+f

⑤ (A+((B-C)*D) ↑ (E+F))

⑥ a * b/c + e/f * g + K -x * y

Binary exp^n tree from Postfix

Postfix $\Rightarrow$ Exp^n tree $\Rightarrow$ Stack

$\Rightarrow$ Scan L $\rightarrow$ R

$\Rightarrow$ Operand $\rightarrow$ push

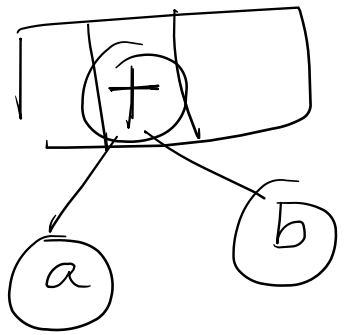
$\Rightarrow$ Operator $\rightarrow$ pop 2 element and construct tree then push the pointer into the stack (eg:- pointer of ⊕)

$\Rightarrow$ **a b +**

L $\rightarrow$ R

eg: | b | a | — 2 times

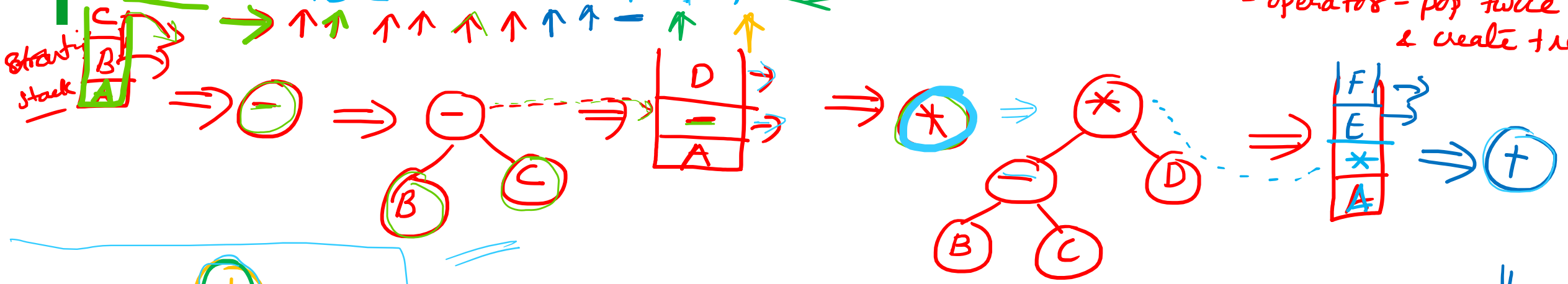
⊕ $\Rightarrow$ pop twice $\Rightarrow$

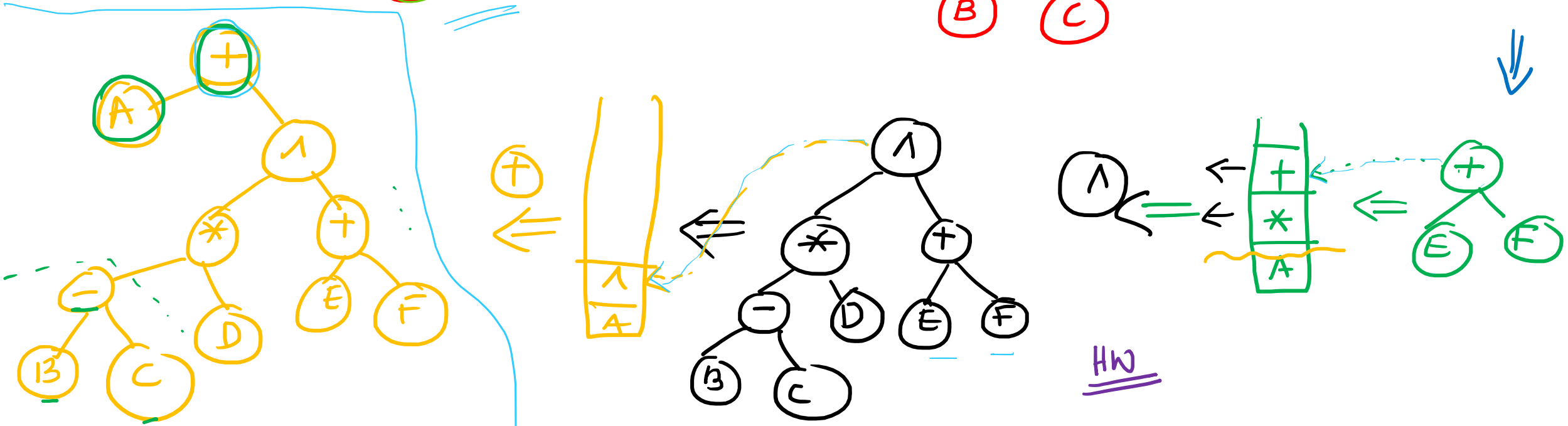


BINARY EXP. TREE from postfix expn.

- Stack - L → R
- operand → push
- operator - pop twice & create tree

Postfix :- A B C - D * E F + ^ +





Final expn tree

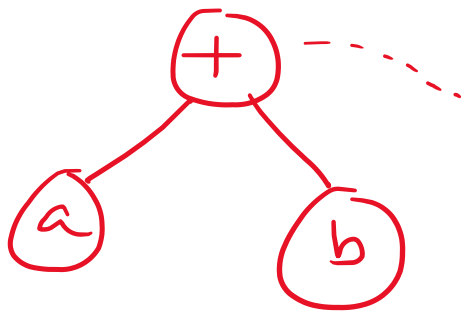
(LRP) ⇒ A B C - D * E F + ^ +

HW

Binary exp^n tree from Preorder

⇒ +ab ←

a
b ⇒ ⊕ ⇒ tree



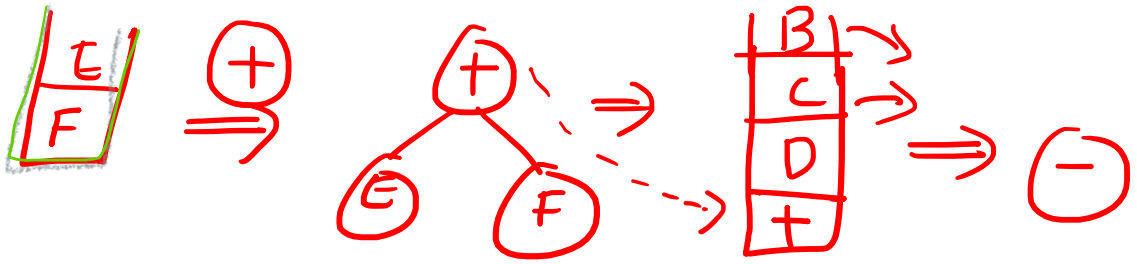
push the pointer for + into stack

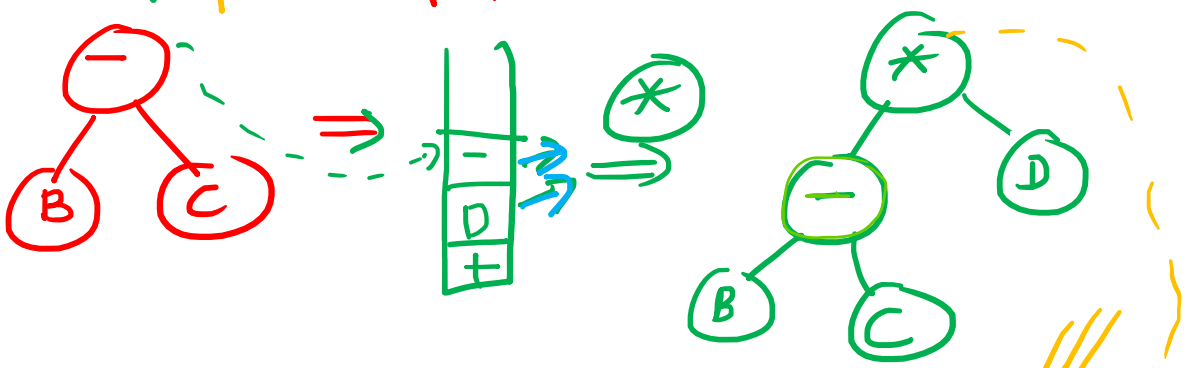
→ Stack DS
→ R → L
→ operands → push
→ operators → pop 2 times & construct the tree

PLR

Binary exp'n tree from Preorder ⇒ + A ∧ * − B C D + E F

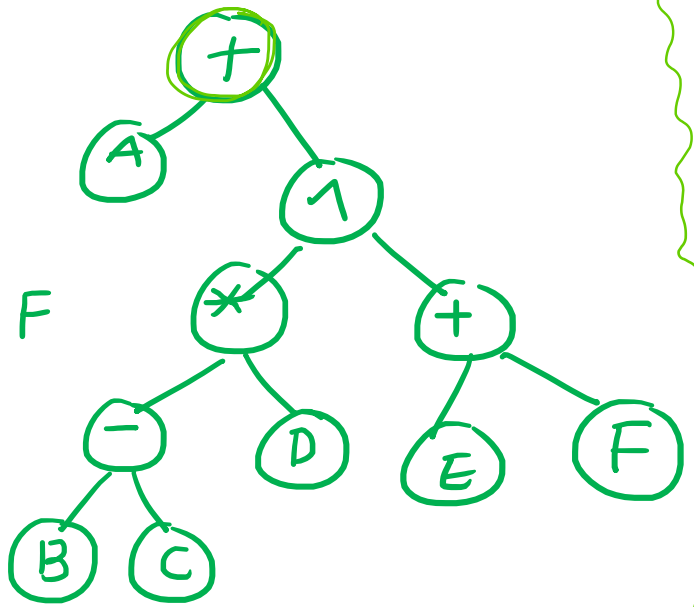
R → L Stack

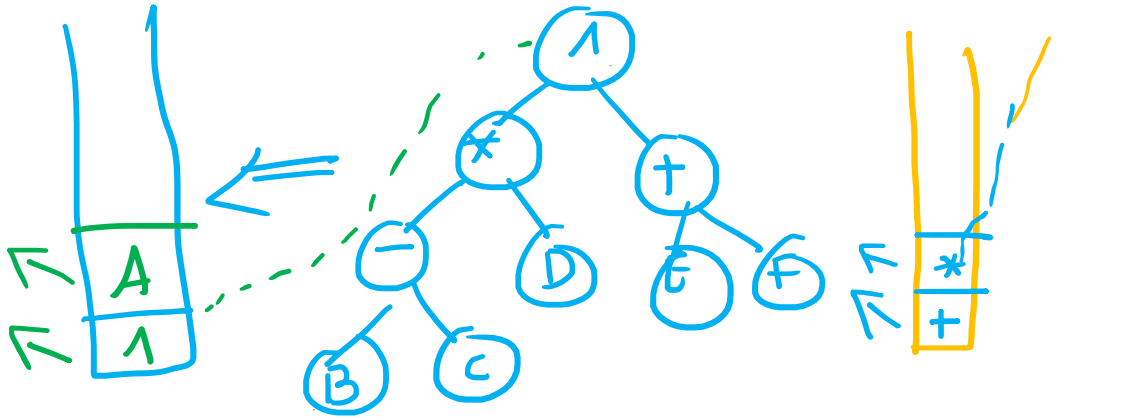




Preorder (PLR)

+ A ∧ * − B C D + E F





HW Pre ⇒ / * + ab − c d + e f

⇒ Post ⇒ ab + cd − * ef + /

Binary Trees

Expression Tree

- Expression tree is a binary tree with the following properties:
  - Each leaf is an operand
  - The root and internal nodes are operators
  - Sub-trees are sub-expressions with the root being an operator

Binary Trees

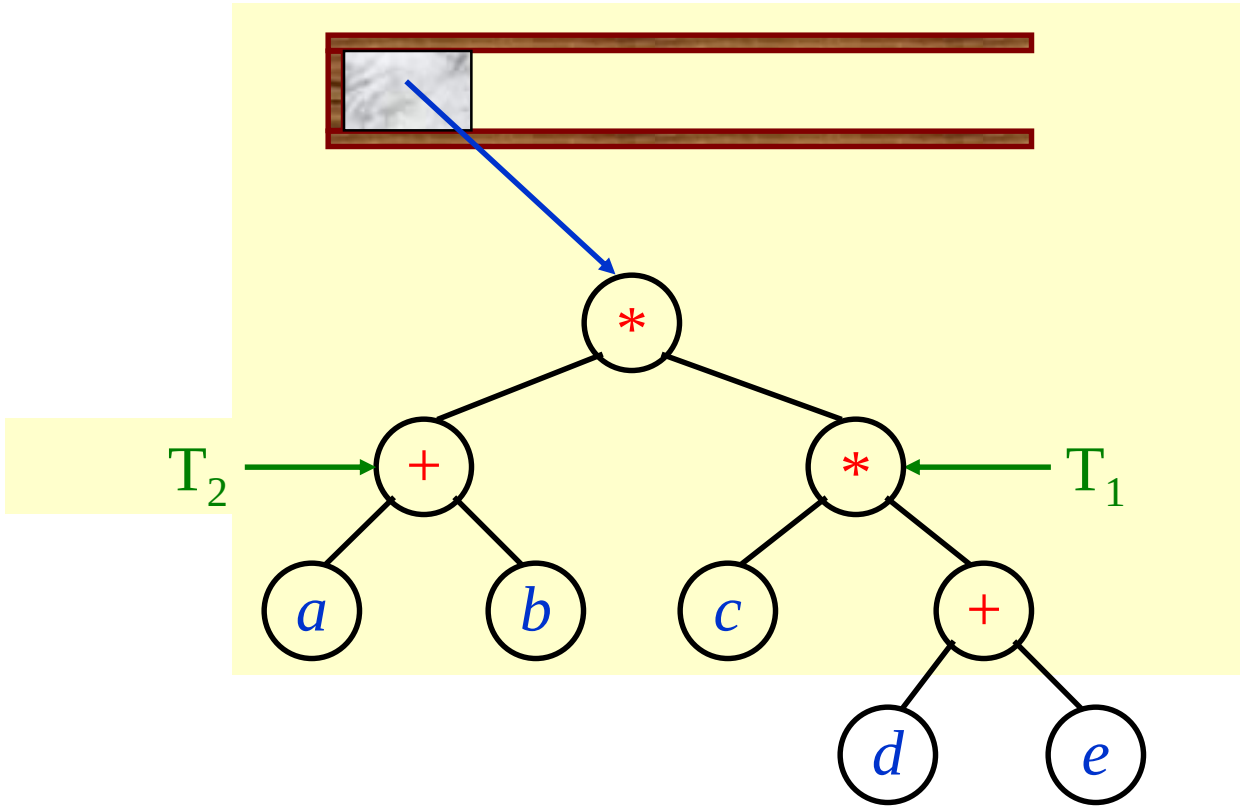
- **Expression Tree** (Continued…)
  - Traversals:
    - Preorder traversal – prefix expression
    - Postorder traversal – postfix expression

Example: (A + ((B – C) * D) ^ (E + F))

- Preorder: +A^*-BCD+EF
- Postorder: ABC-D*EF+^+

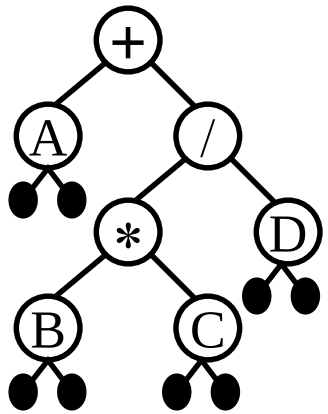
Expression Trees (syntax trees)

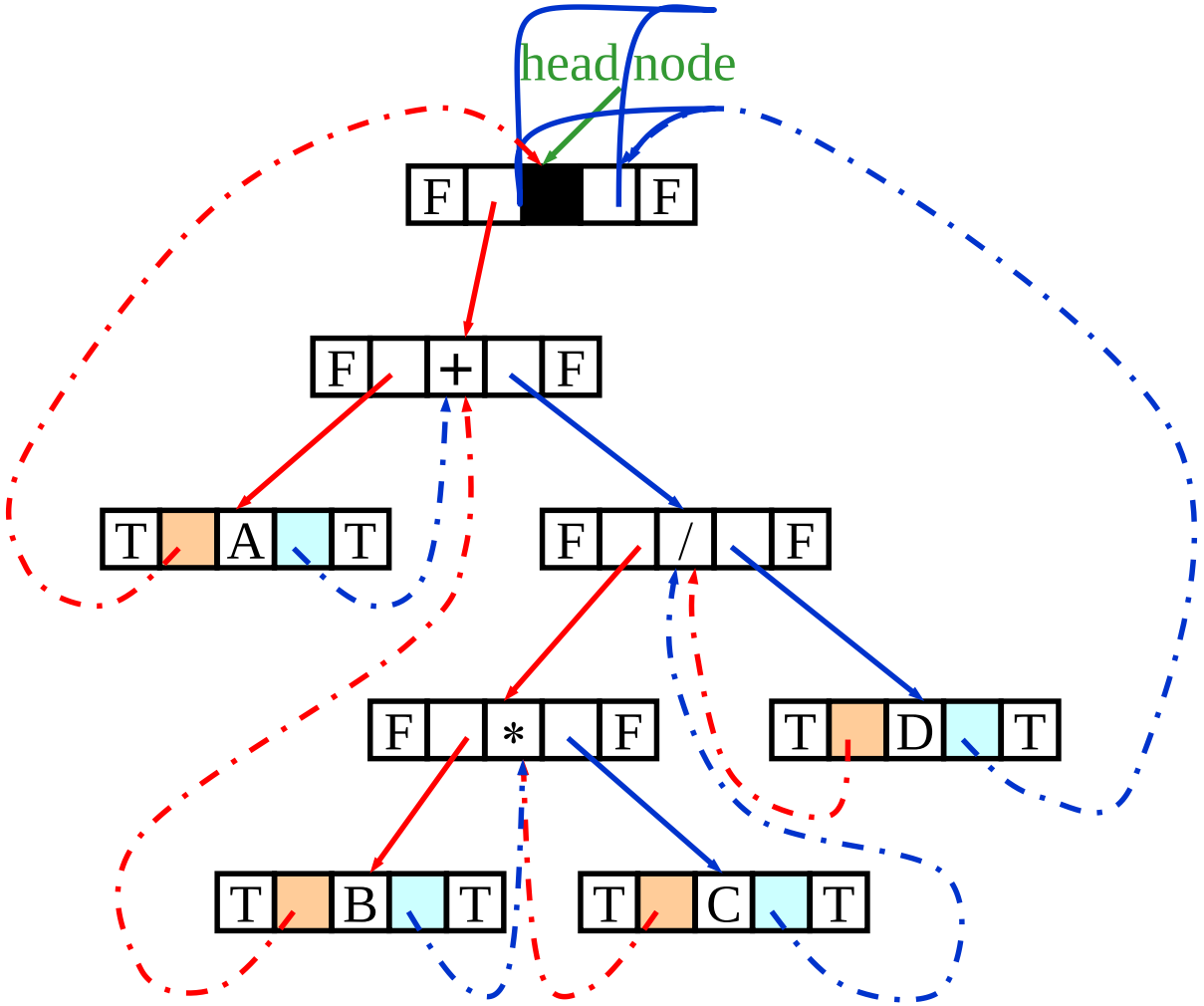
〖Example〗 (*a* + *b*) * (*c* * (*d* + *e*)) = *a b* + *c d e* + * *



〖Example〗 Given the syntax tree of an expression (infix)

$A + B * C / D$





```
a x (b + c) + d
```

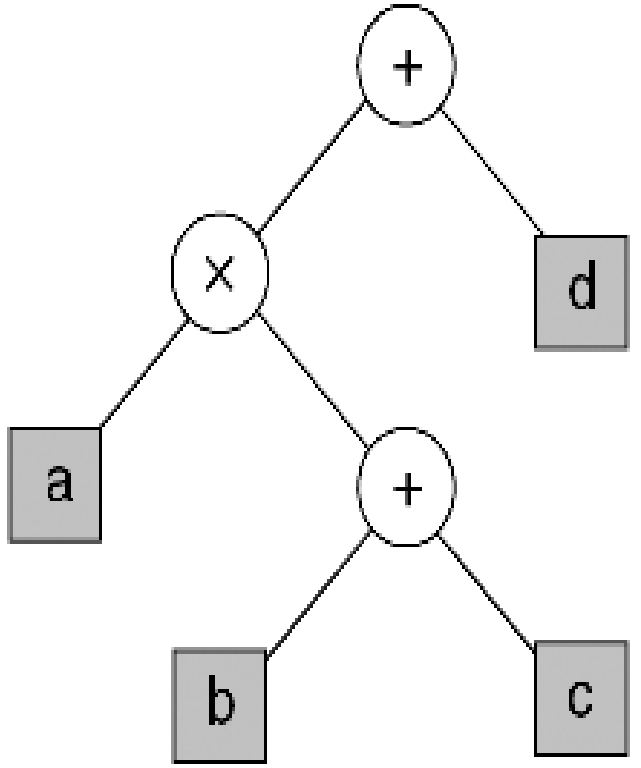
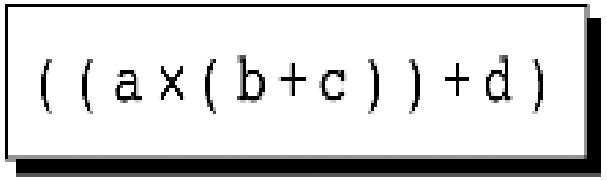


FIGURE 6-15 Infix Expression and Its Expression Tree



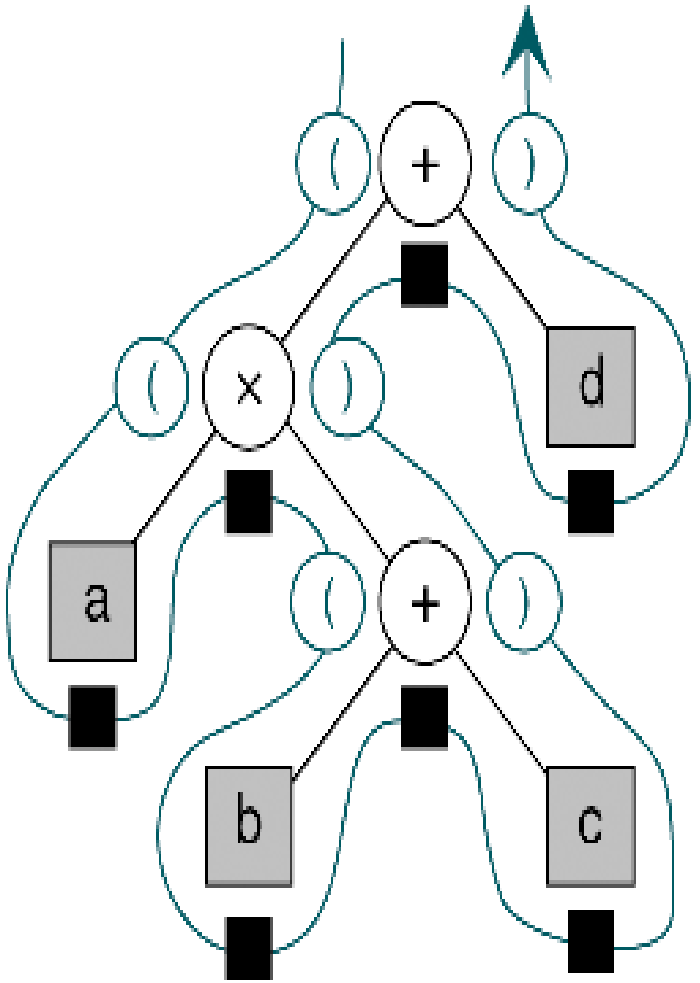
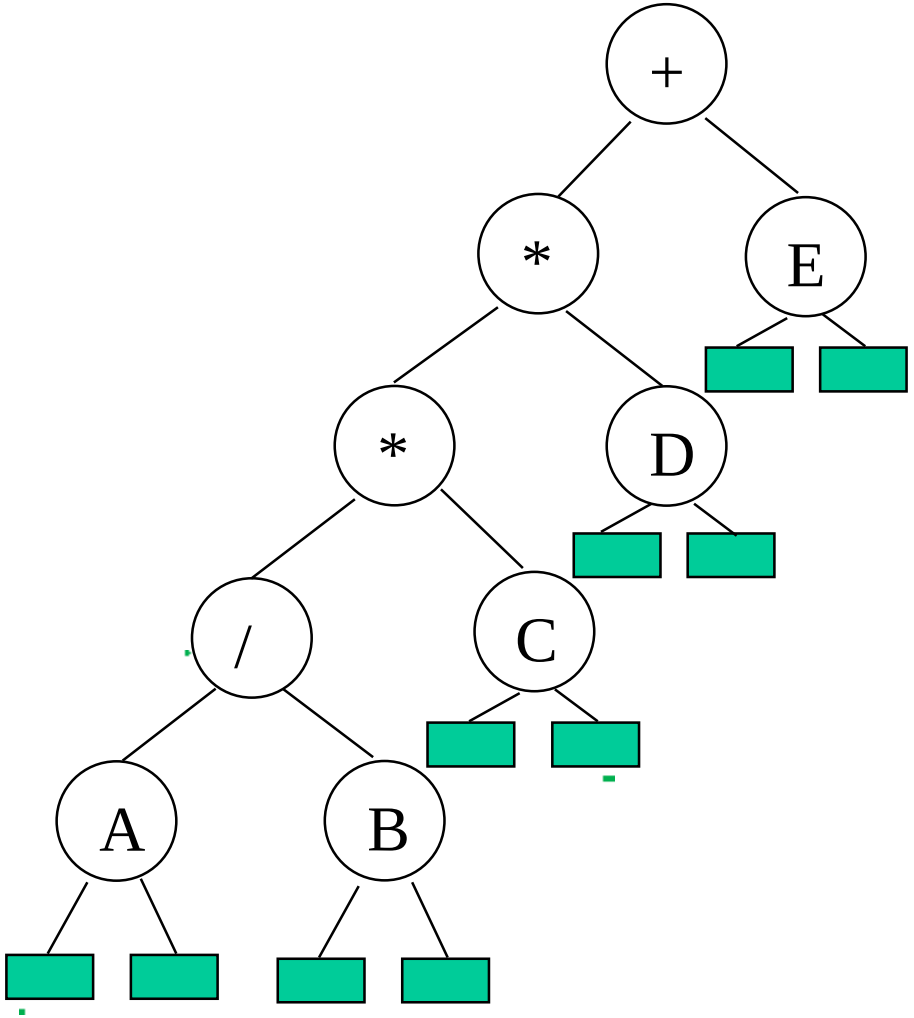


FIGURE 6-16 Infix Traversal of an Expression Tree

Arithmetic Expression Using BT



inorder traversal
A / B * C * D + E
infix expression
preorder traversal
+ * * / A B C D E
prefix expression
postorder traversal
A B / C * D * E +
postfix expression
level order traversal
+ * E * D / C A B

ALGORITHM 6-6 Infix Expression Tree Traversal

```
Algorithm infix (tree)
Print the infix expression for an expression tree.
   Pre  tree is a pointer to an expression tree
   Post the infix expression has been printed
 1 if (tree not empty)
   1  if (tree token is an operand)
      1  print (tree-token)
   2  else
      1  print (open parenthesis)
      2  infix (tree left subtree)
      3  print (tree token)
      4  infix (tree right subtree)
      5  print (close parenthesis)
   3  end if
 2 end if
end infix
```

ALGORITHM 6-7 Postfix Traversal of an Expression Tree

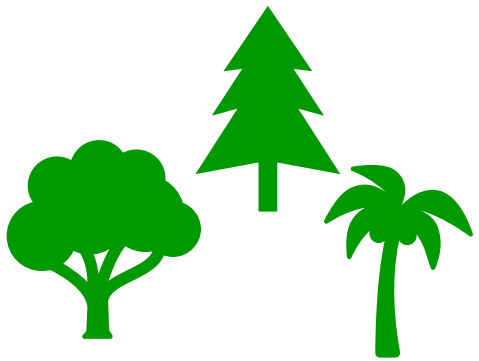
```
Algorithm postfix (tree)
Print the postfix expression for an expression tree.
   Pre  tree is a pointer to an expression tree
   Post the postfix expression has been printed
 1 if (tree not empty)
      1   postfix (tree left subtree)
      2   postfix (tree right subtree)
      3   print   (tree token)
 2 end if
end postfix
```

ALGORITHM 6-8 Prefix Traversal of an Expression Tree

```
Algorithm prefix (tree)
Print the prefix expression for an expression tree.
   Pre  tree is a pointer to an expression tree
   Post the prefix expression has been printed
1  if (tree not empty)
   1   print  (tree token)
   2   prefix (tree left subtree)
   3   prefix (tree right subtree)
2  end if
end prefix
```

Types of Trees

- Binary Trees
- Binary Search Trees



Binary Search Trees.

The ordered nodes.

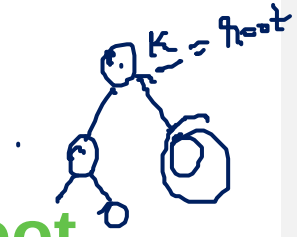
Objectives.

- Create and implement binary search trees
- Understand the operation of the binary search tree ADT
- Write application programs using the binary search tree ADT
- Design and implement a list using a BST

Basic Concepts.

- **Binary search trees** provide an excellent structure for searching a list and at the same time for inserting and deleting data into the list.

- A **binary search tree (BST) is a binary tree** with following properties:
  - All items in the **left of the tree** are **less than the root**.
  - All items in the **right subtree** are **greater than or equal to the root**.
  - **Each subtree** is itself a **binary search tree**.



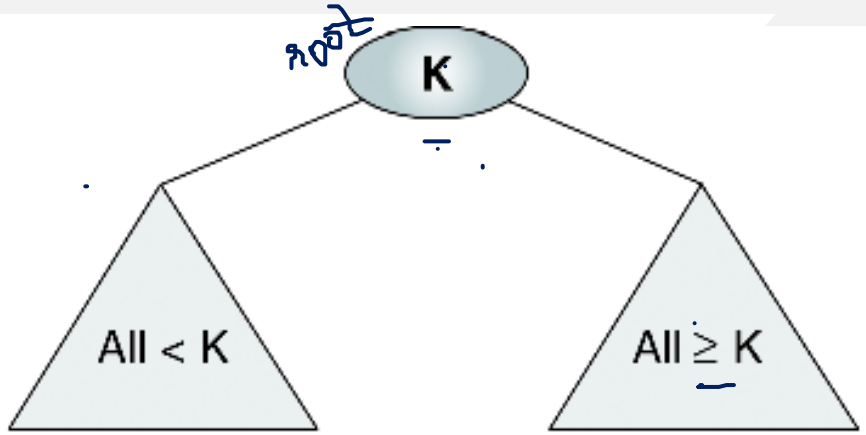


FIGURE 7-1 Binary Search Tree

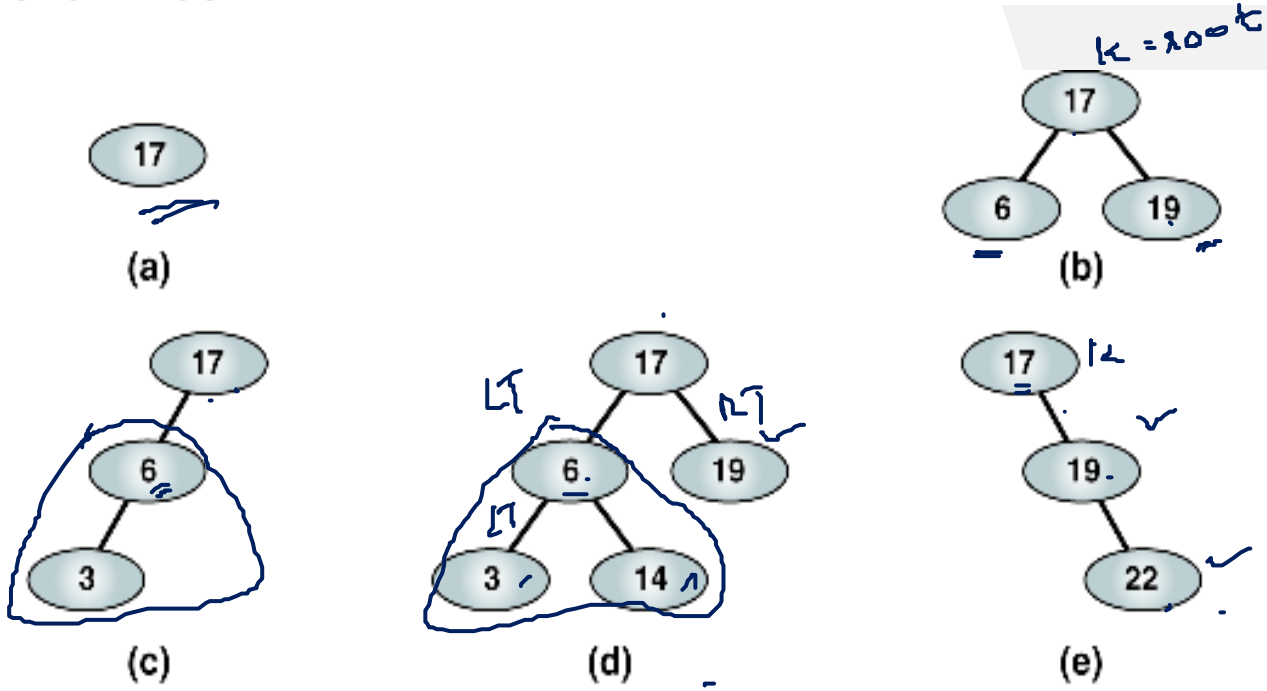


FIGURE 7-2 Valid Binary Search Trees

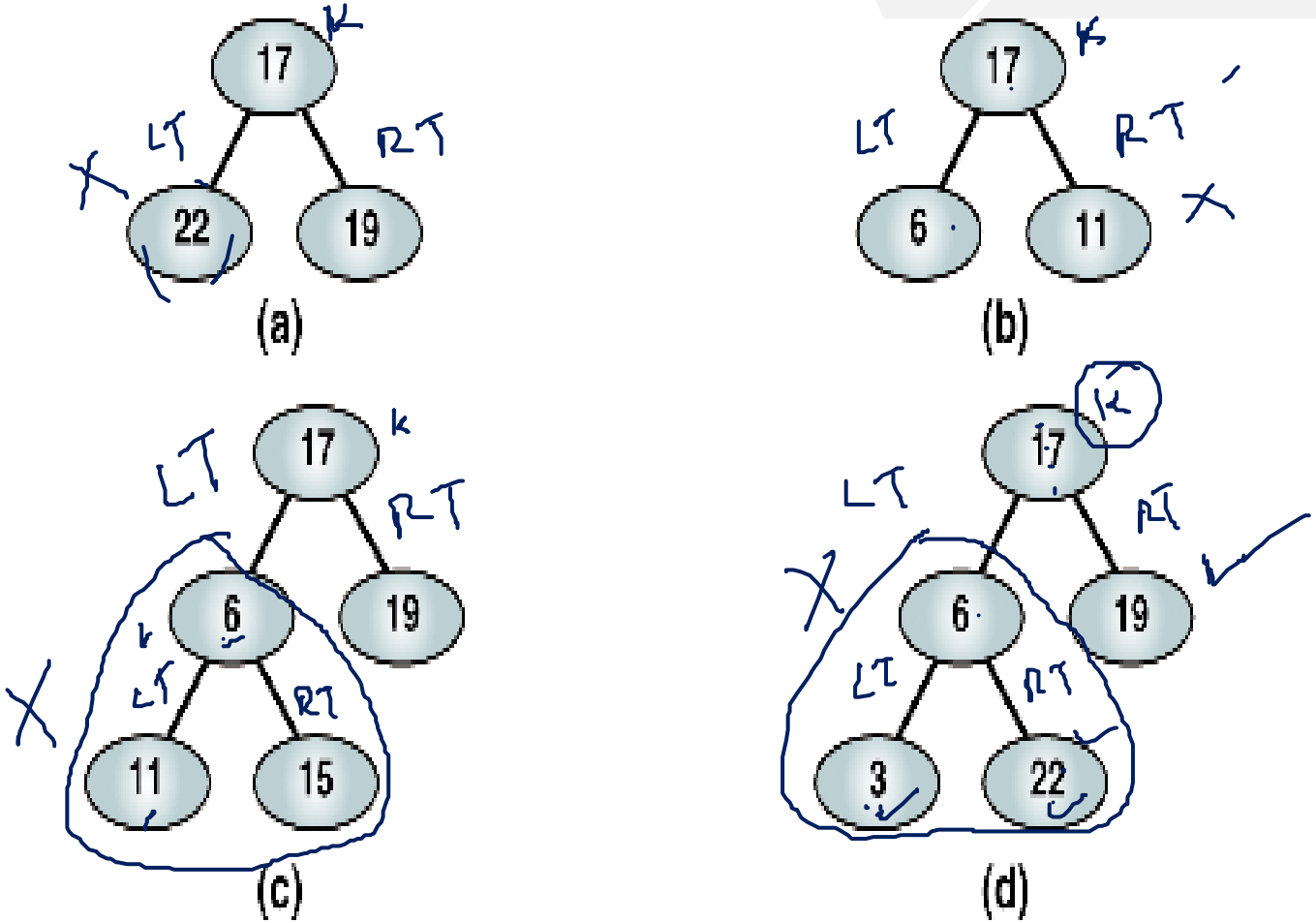


FIGURE 7-3 Invalid Binary Search Trees

BST OPERATIONS.

There are ***four*** basic BST operations:

- **Traversal**
- **Search** = key element
- **Insert, and**
- **Delete.**

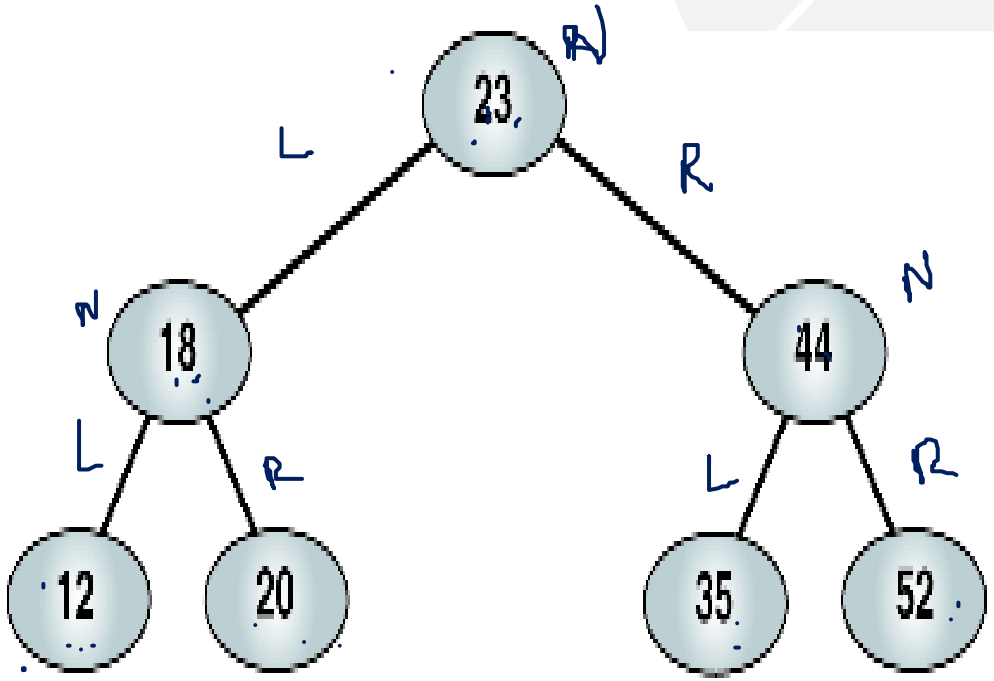


FIGURE 7-4 Example of a Binary Search Tree

Preorder Traversal : 23 18 12 20 44 35 52

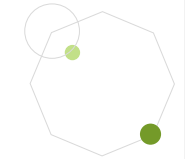
Inorder Traversal: 12 18 20 23 35 44 52

Postorder Traversal: 12 20 18 35 52 44 23

SEARCH OPERATIONS.

Three search algorithms:

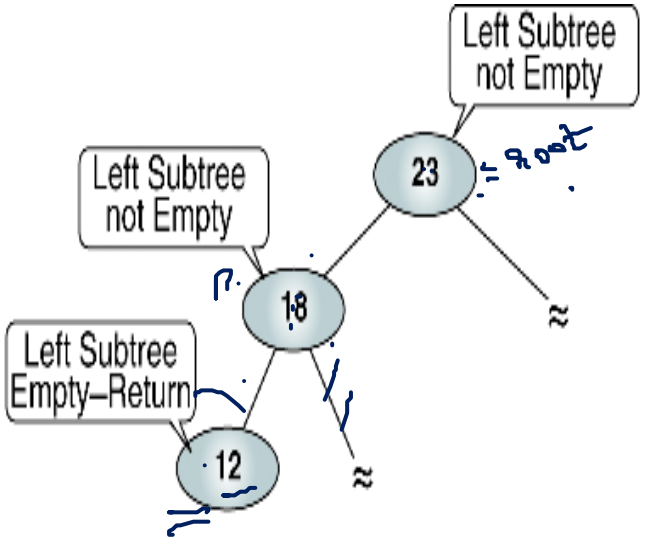
- find the smallest node
- find the largest node
- find a requested node (BST search).



ALGORITHM 7-1 Find Smallest Node in a BST

Find the smallest node

```
Algorithm findSmallestBST (root)
This algorithm finds the smallest node in a BST.
   Pre    root is a pointer to a nonempty BST or subtree
   Return address of smallest node
1  if (left subtree empty)
   1   return (root)
2  end if
3  return findSmallestBST (left subtree)
end findSmallestBST
```

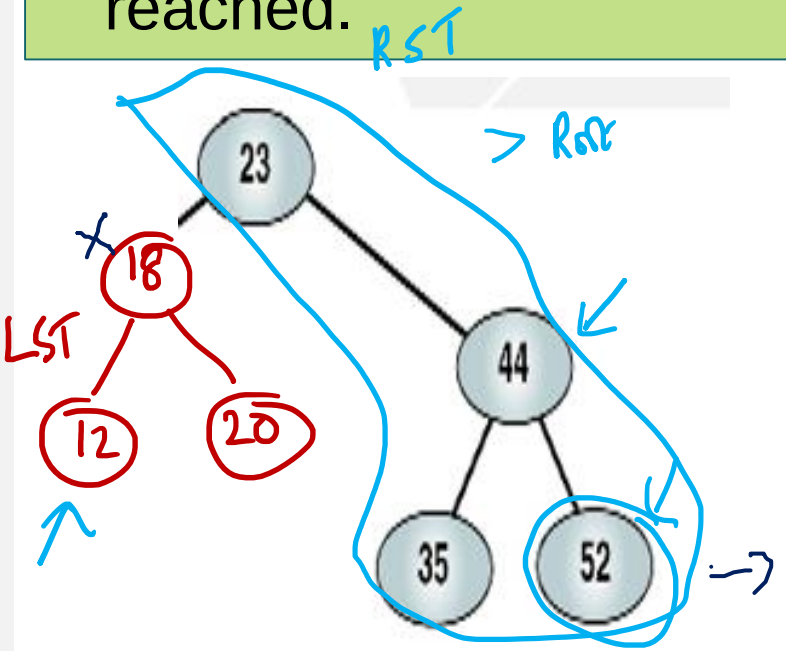


- Node with the smallest value (12) is the ***far-left leaf node*** in the tree.
- The find smallest node operation, therefore, simply follows the left branches until a leaf is reached..

FIGURE 7-5 Find Smallest Node in a BST

ALGORITHM 7-2 Find Largest Node in a BST

- Node with the largest value (52) is the ***far-right leaf node*** in the tree.
- This operation, therefore, simply follows the right branches until a leaf is reached.




```
Algorithm findLargestBST (root)
This algorithm finds the largest node in a BST.
   Pre    root is a pointer to a nonempty BST or subtree
   Return address of largest node returned
1  if (right subtree empty)
   1   return (root)
2  end if
3  return findLargestBST (right subtree)
end findLargestBST
```

Find the largest node.

binary search algorithm

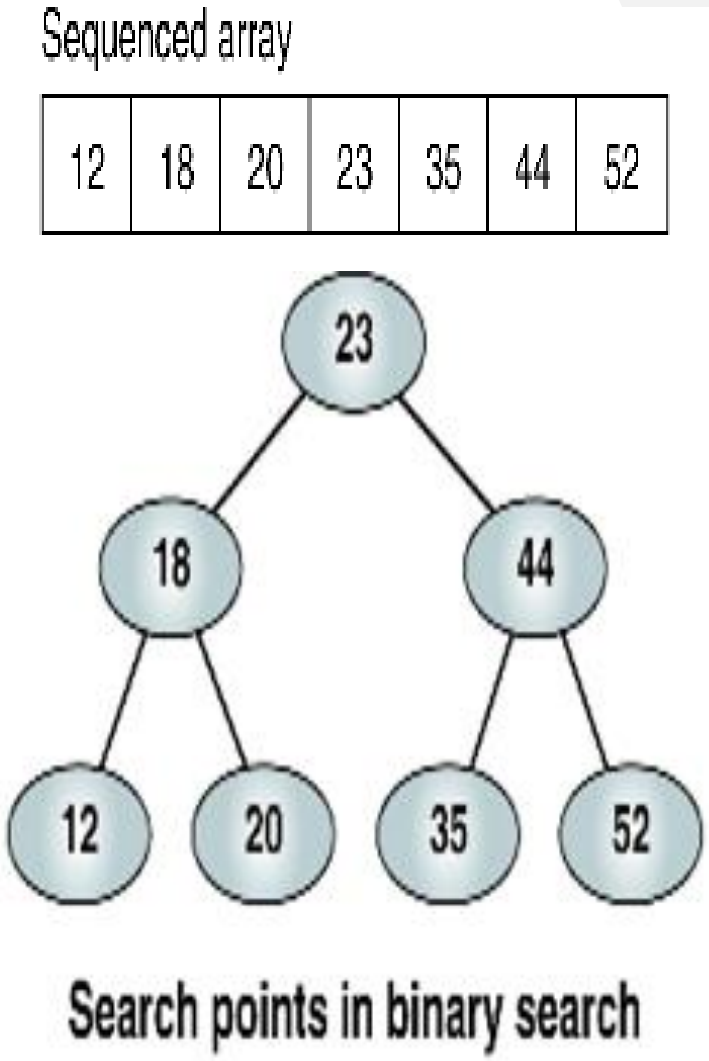
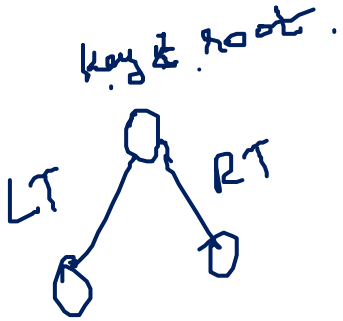


FIGURE 7-6 BST and the Binary Search

ALGORITHM 7-3 Search BST




```
Algorithm searchBST (root, targetKey)
Search a binary search tree for a given value.
   Pre    root is the root to a binary tree or subtree
          targetKey is the key value requested
   Return the node address if the value is found
          null if the node is not in the tree
 1 if (empty tree)
      Not found
   1  return null => no element
 2 end if
 3 if (targetKey < root)
   1  return searchBST (left subtree, targetKey)
 4 else if (targetKey > root)
   1  return searchBST (right subtree, targetKey)
 5 else
      Found target key
   1  return root
 6 end if
end searchBST
```

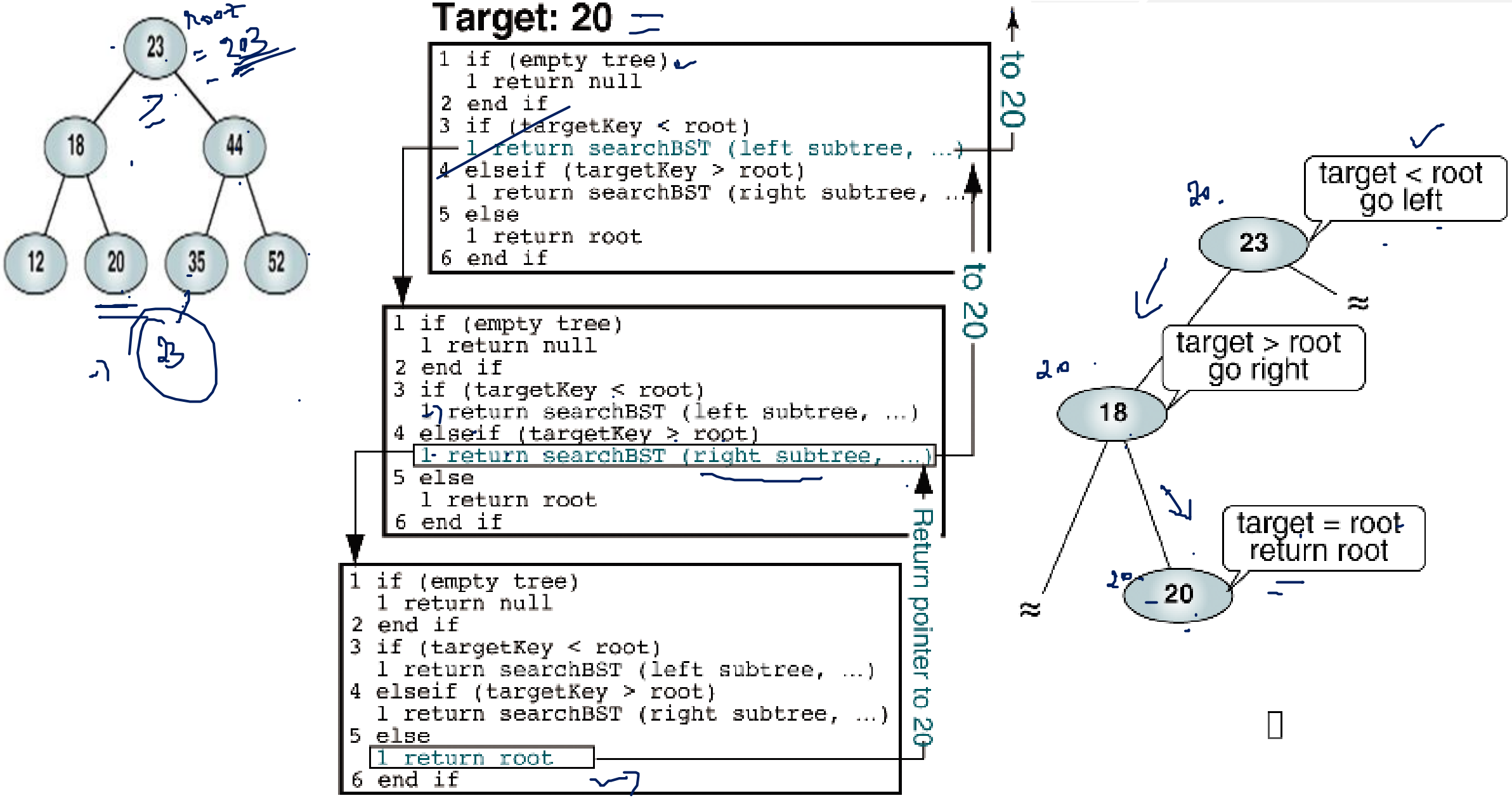
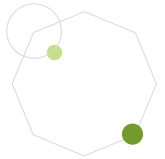


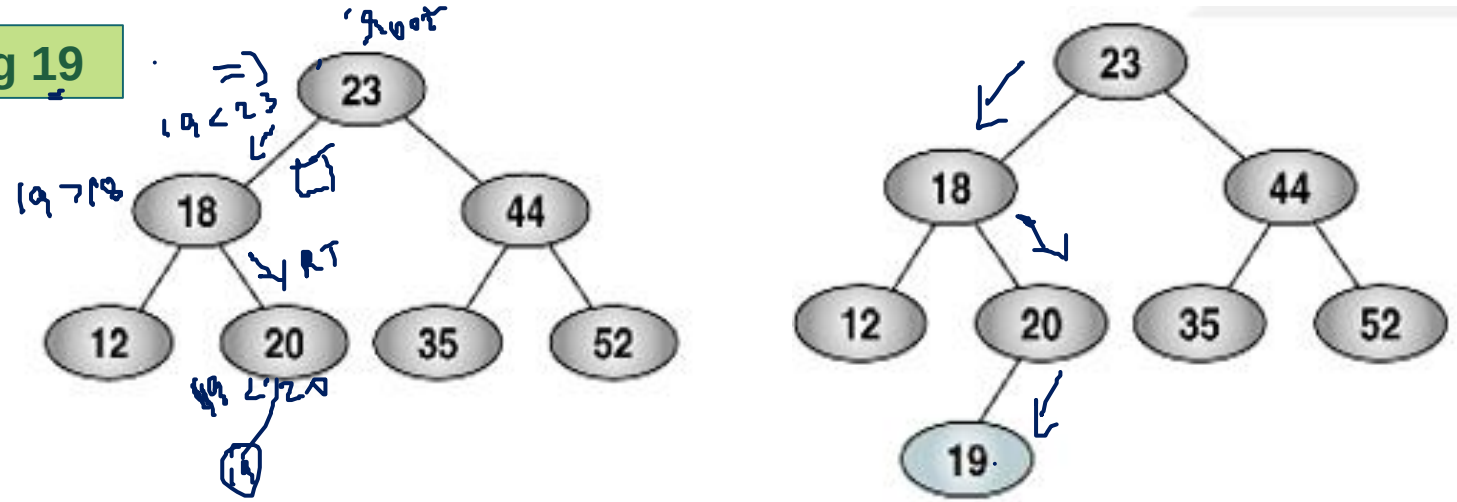
FIGURE 7-7 Searching a BST

INSERTION OPERATIONS.

All BST insertions take place at a leaf or a leaflike node.



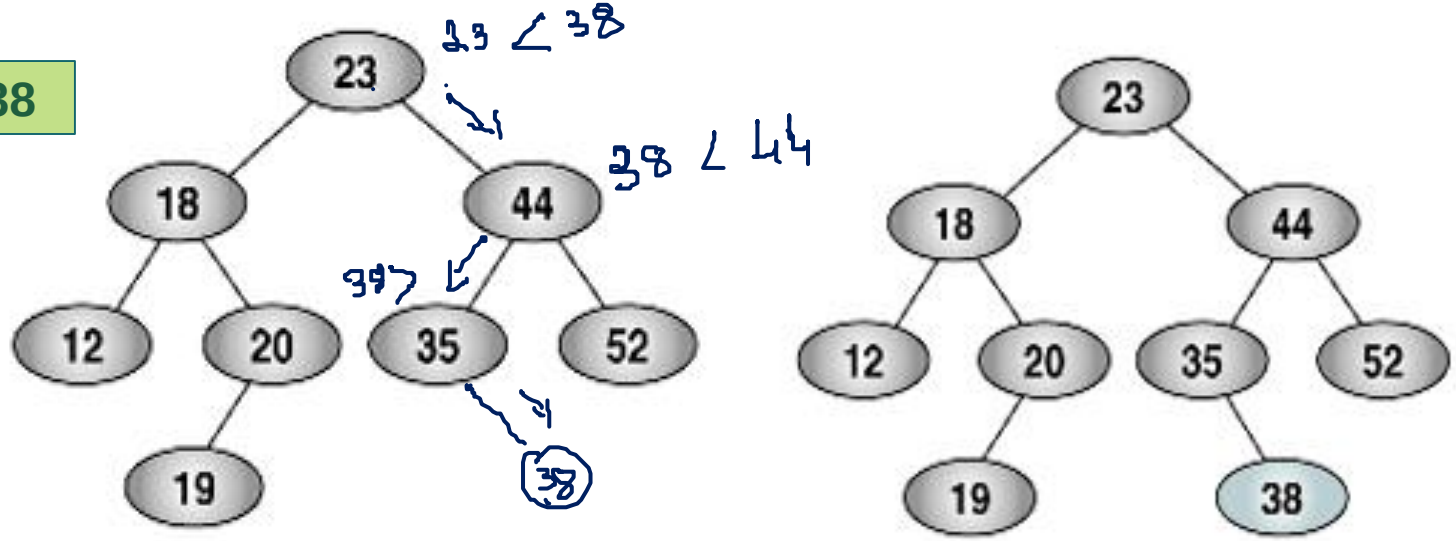
Inserting 19



(a) Before inserting 19

(b) After inserting 19

Inserting 38

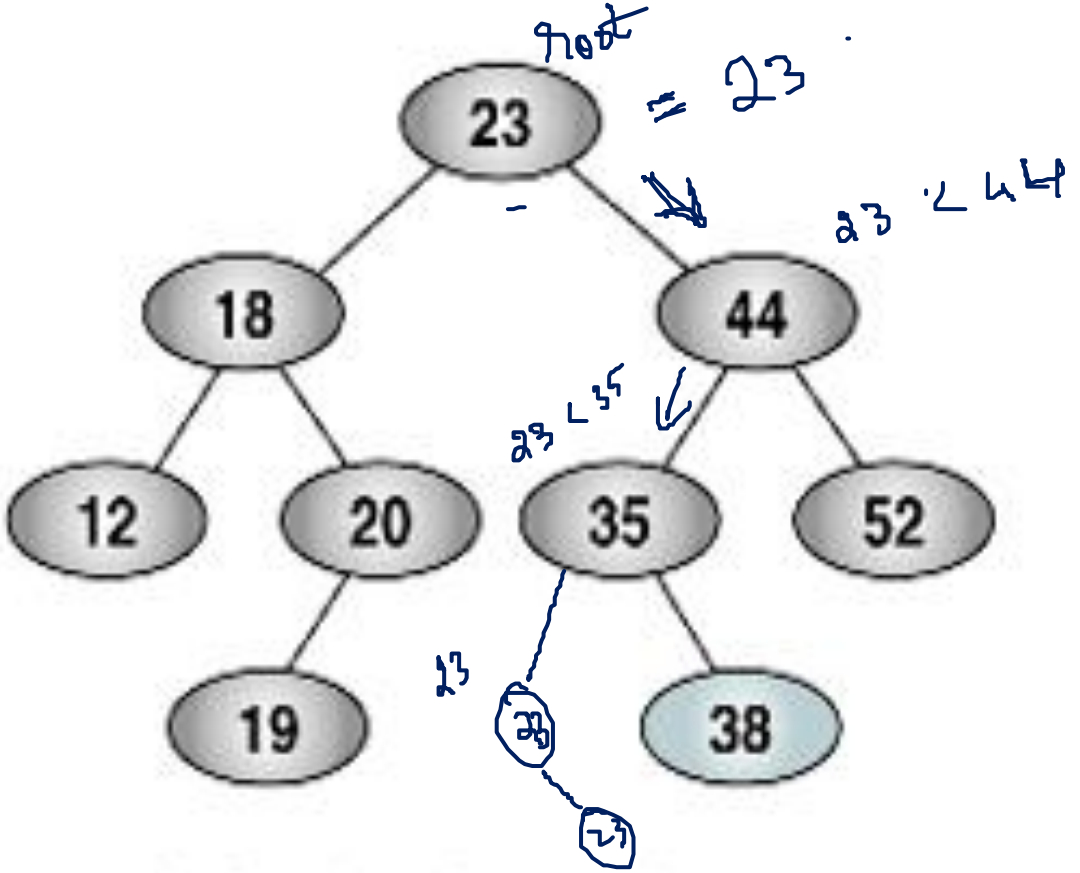


(c) Before inserting 38

(d) After inserting 38

FIGURE 7-8 BST Insertion

Inserting 23 ?



(d) After inserting 38

FIGURE 7-8 BST Insertion

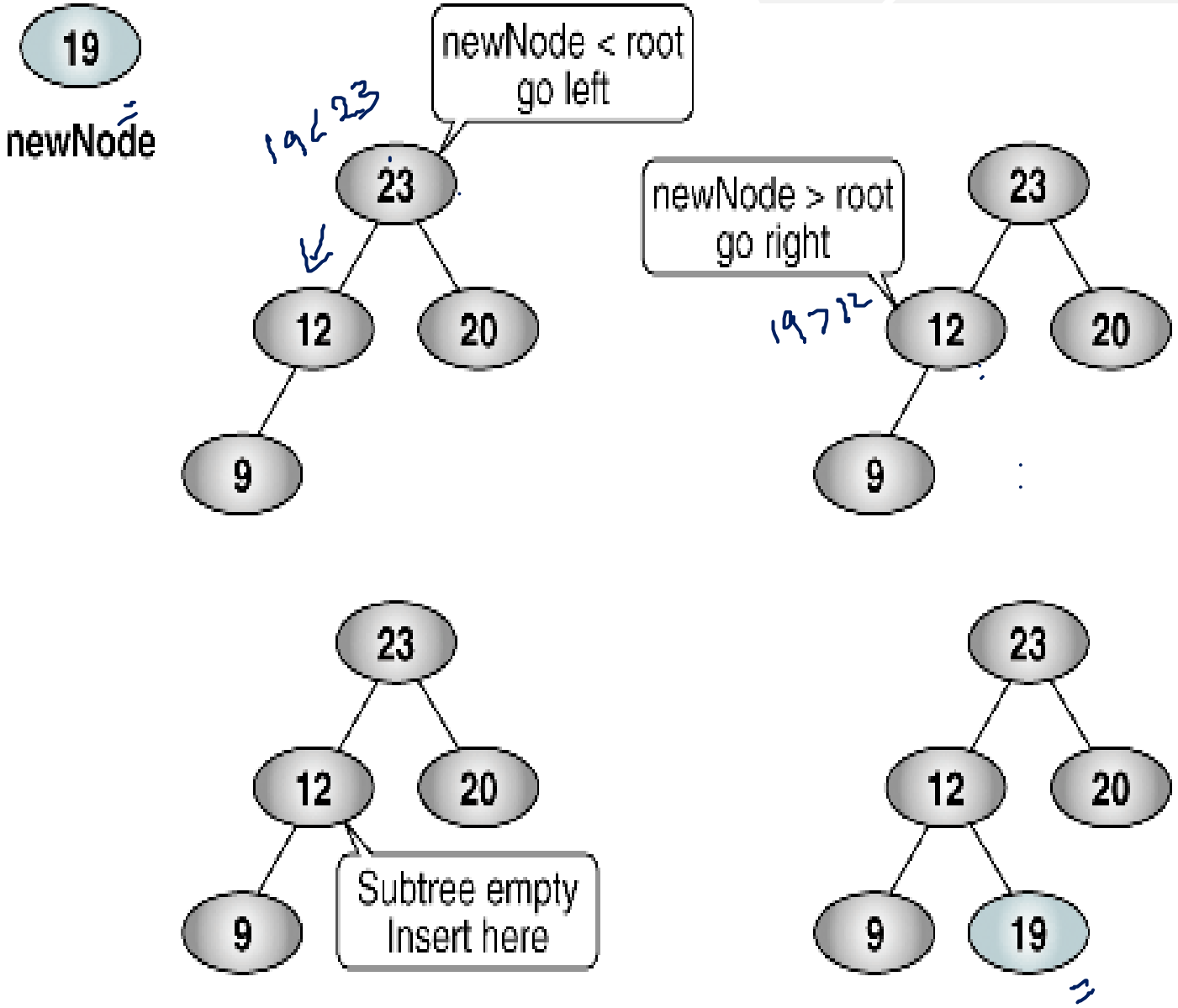


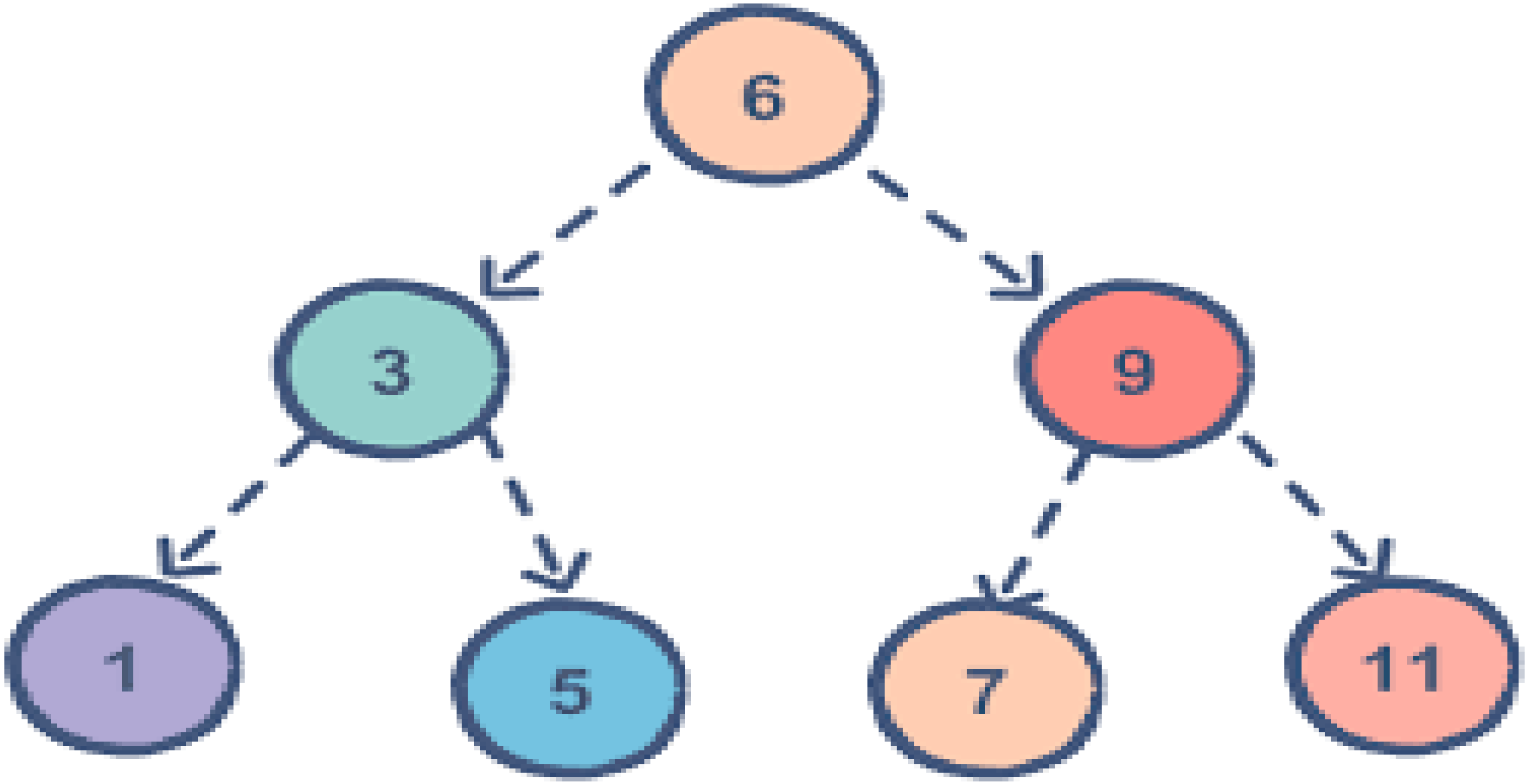
FIGURE 7-9 Trace of Recursive BST Insert

ALGORITHM 7-4 Add Node to BST

```
Algorithm addBST (root, newNode)
Insert node containing new data into BST using recursion.
   Pre    root is address of current node in a BST
          newNode is address of node containing data
   Post   newNode inserted into the tree
   Return address of potential new tree root
1  if (empty tree)
   1   set root to newNode
   2   return newNode
2  end if
```

ALGORITHM 7-4 Add Node to BST *(continued)*

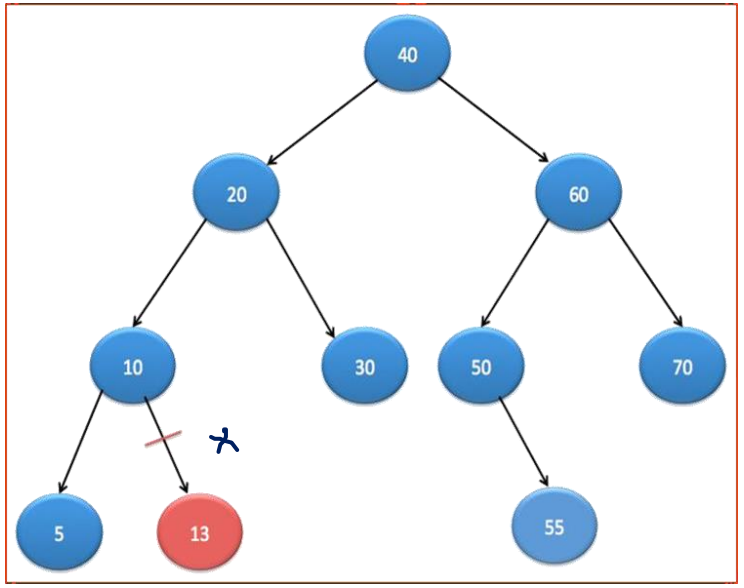
```
   Locate null subtree for insertion
3  if (newNode < root)
   1   return addBST (left subtree, newNode)
4  else
   1   return addBST (right subtree, newNode)
5  end if
end addBST
```



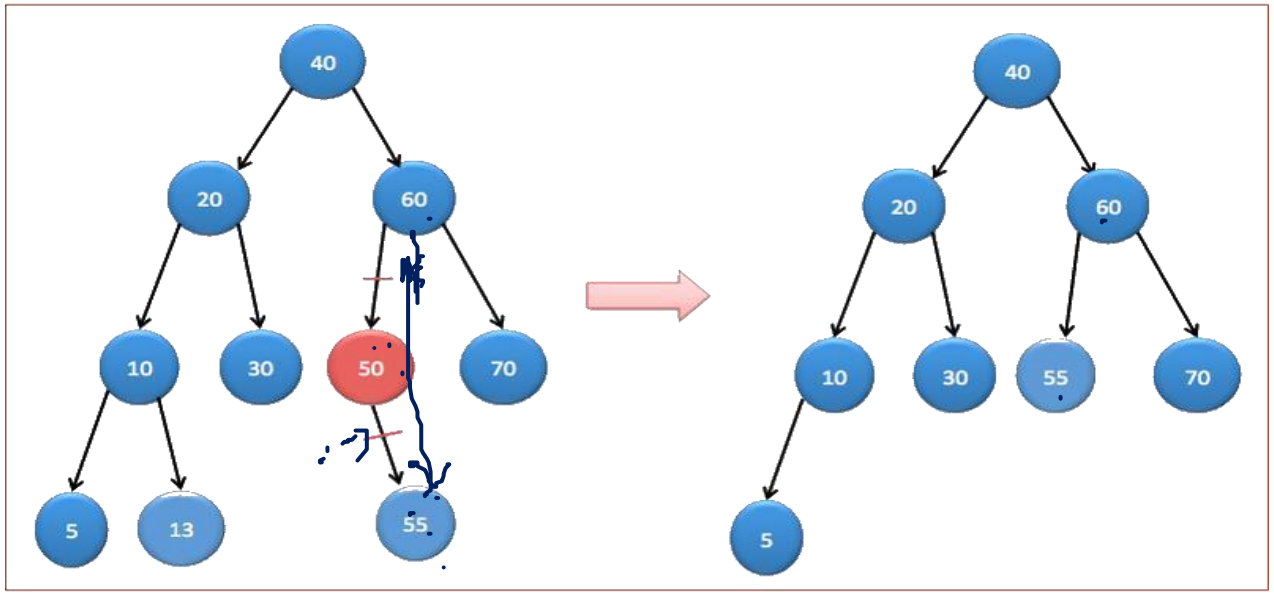
An example of a binary search tree

Deletion.

- To **delete a node** from a binary search tree, we must **first locate it**.
- There are **four cases** for deletion:
  1. The node to be deleted has **no children,** which means ***delete the node/ leaf node.***
  2. The node to be deleted has **only a right subtree**. Delete the node and **attach the right subtree** to the deleted node's parent.



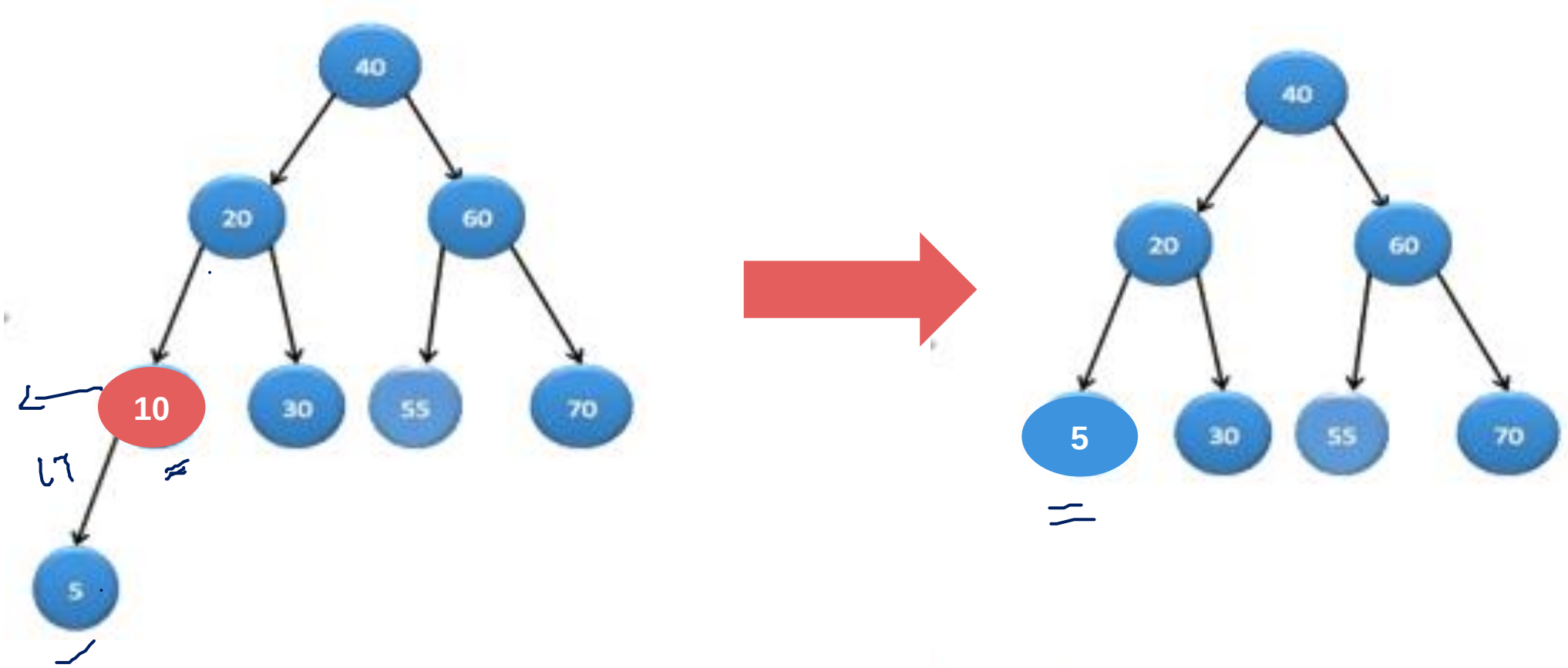
No children



Node with right subtree

Deletion.

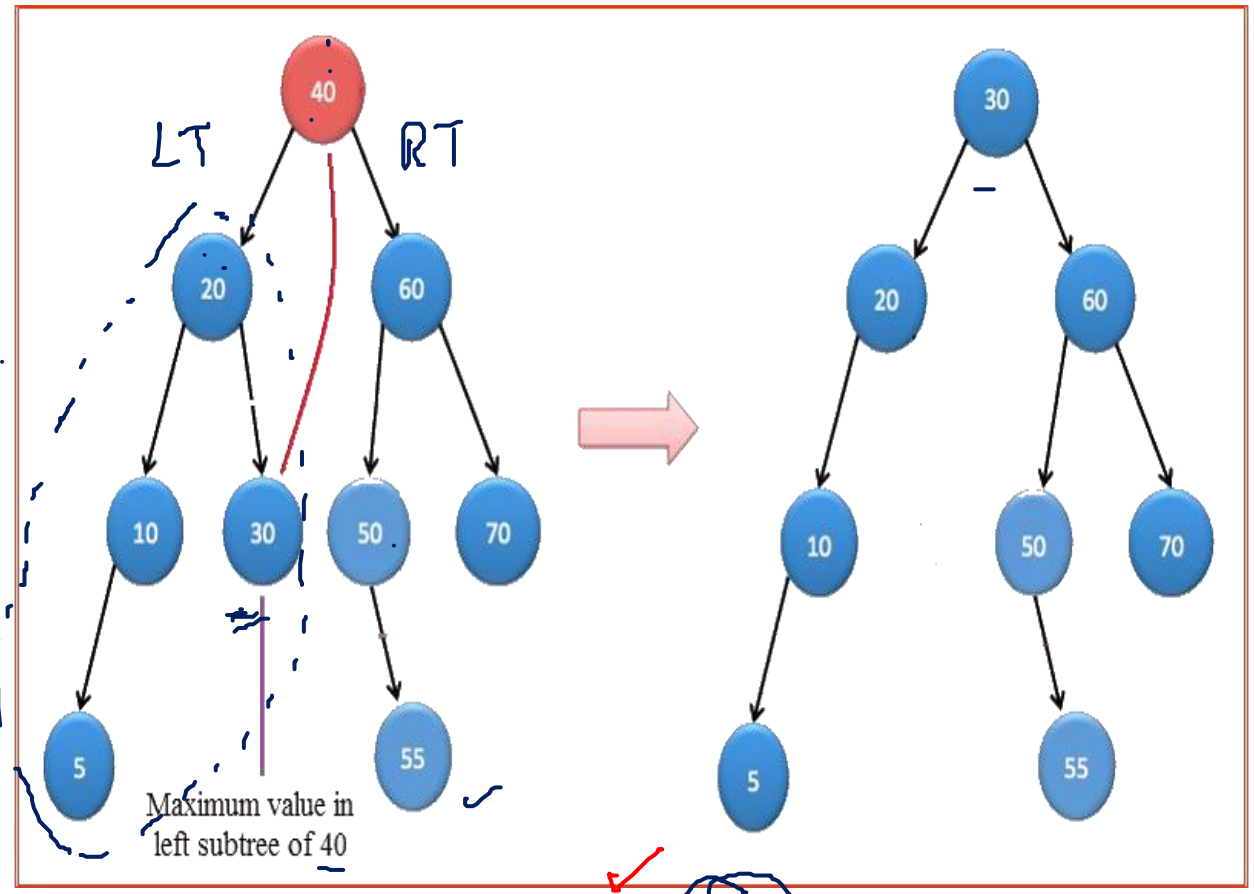
3. The node to be deleted has **only a left subtree**. Delete the node and **attach the left subtree** to the deleted node's parent.

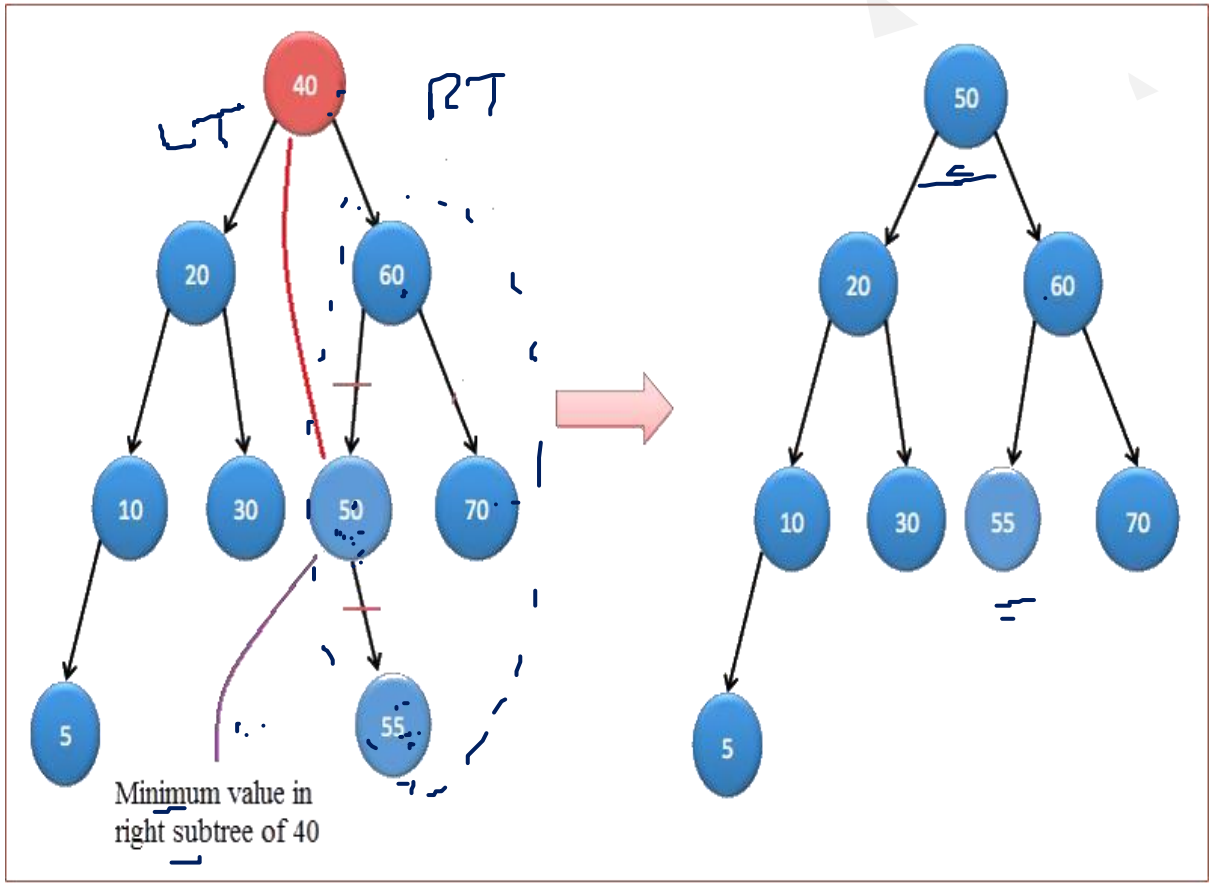


Node with left subtree

Deletion.

4. The node to deleted has **two subtrees**. Tree becomes unbalanced.





=> IN => 5 10 20 30 40 50 55 60 70

LT RT

Node with two subtrees

Deletion of node with two subtrees – two ways

- When the node to deleted **has two subtrees.**
- Find the **largest node in the deleted node's left subtree** and **move its data to replace the deleted node's data**.
- Do the **inorder traversal** -> **5, 10, 20, 30, 40, 50, 55, 60, 70**.
- Find inorder predecessor of root, replace 40 by 30.

Or

- Find the **smallest node in the deleted node's right subtree** and **move its data to replace the deleted node's data**.
- Do the **inorder traversal** -> **5, 10, 20, 30, 40, 50, 55, 60, 70**.
- Find inorder successor of root, replace 40 by 50.

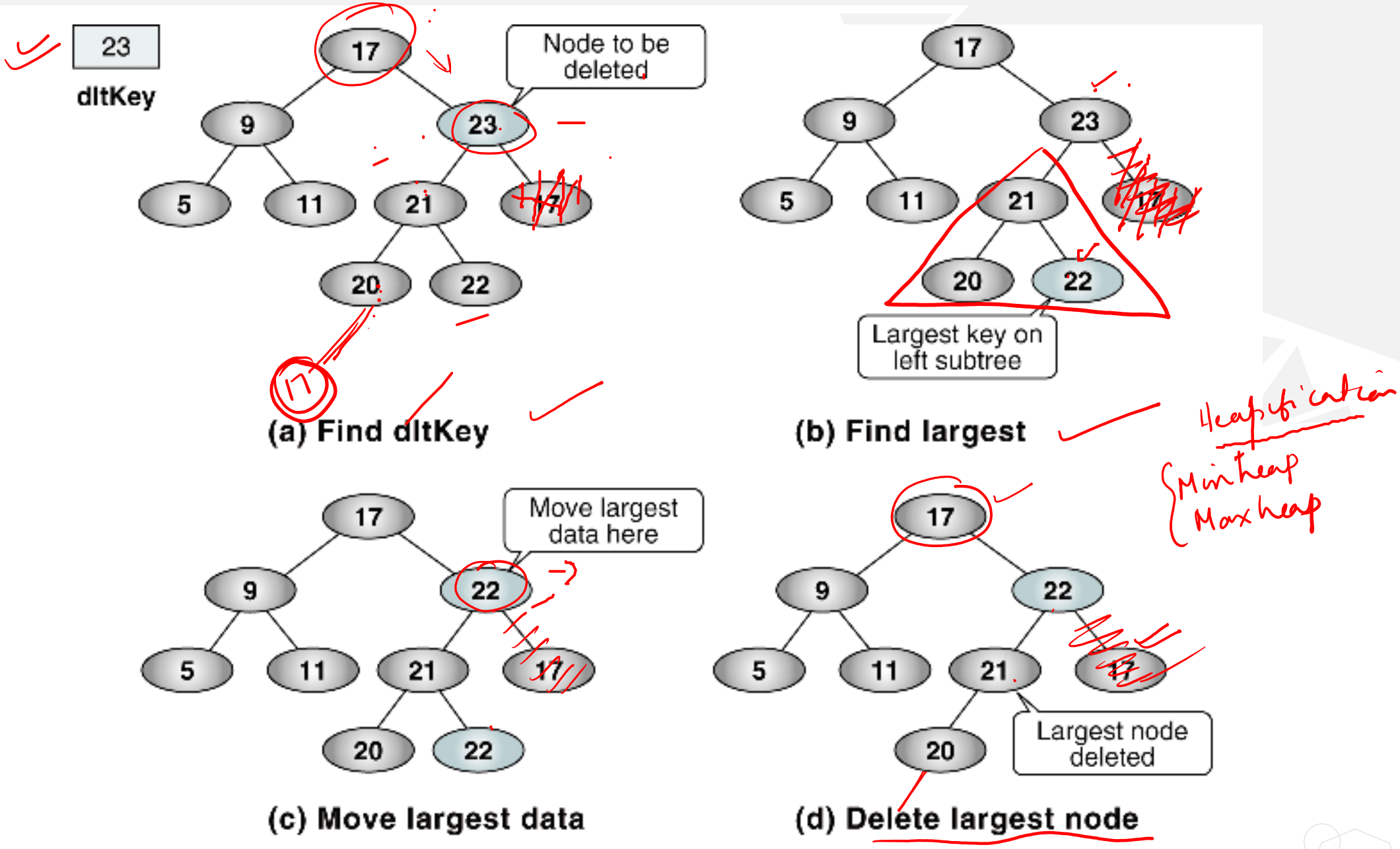
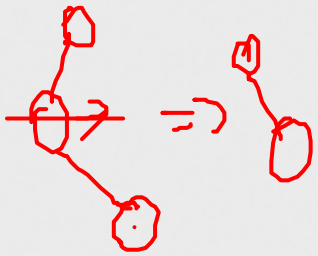


FIGURE 7-10 Delete BST Test Cases

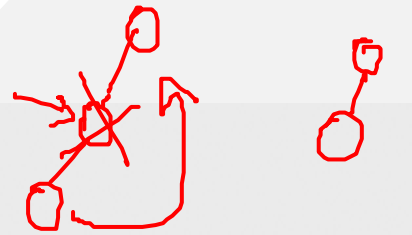
ALGORITHM 7-5 Delete Node from BST

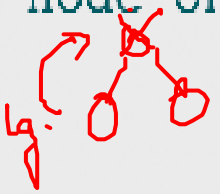
```
Algorithm deleteBST (root, dltKey)
This algorithm deletes a node from a BST.
   Pre    root is reference to node to be deleted
          dltKey is key of node to be deleted
   Post   node deleted
          if dltKey not found, root unchanged
   Return true if node deleted, false if not found
1  if (empty tree)
   1   return false
2  end if
3  if (dltKey < root)
   1   return deleteBST (left subtree, dltKey)
4  else if (dltKey > root)
   1   return deleteBST (right subtree, dltKey)
5  else
       Delete node found--test for leaf node
   1   If (no left subtree)
       1   make right subtree the root
       2   return true
```



ALGORITHM 7-5 Delete Node from BST (continued)

```
   2   else if (no right subtree)
       1   make left subtree the root
       2   return true
   3   else
           Node to be deleted not a leaf. Find largest node on
           left subtree.
       1   save root in deleteNode
       2   set largest to largestBST (left subtree)
       3   move data in largest to deleteNode
       4   return deleteBST (left subtree of deleteNode,
                             key of largest
   4   end if
6  end if
end deleteBST
```



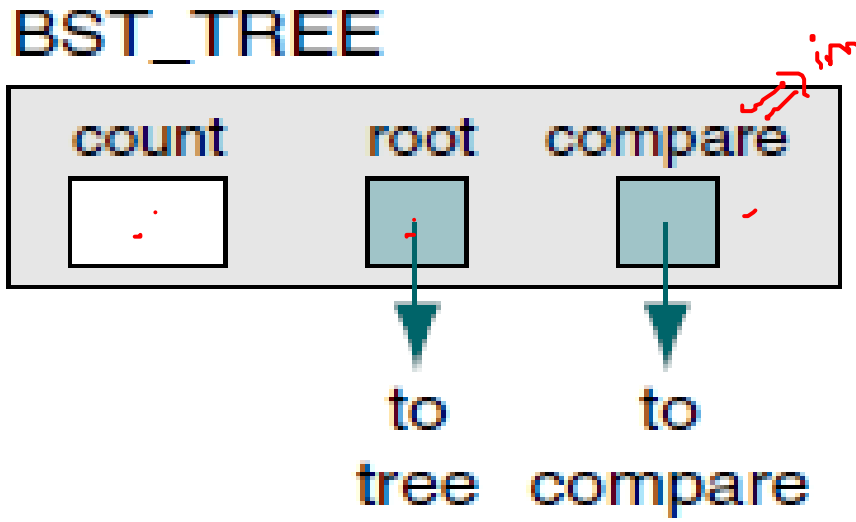


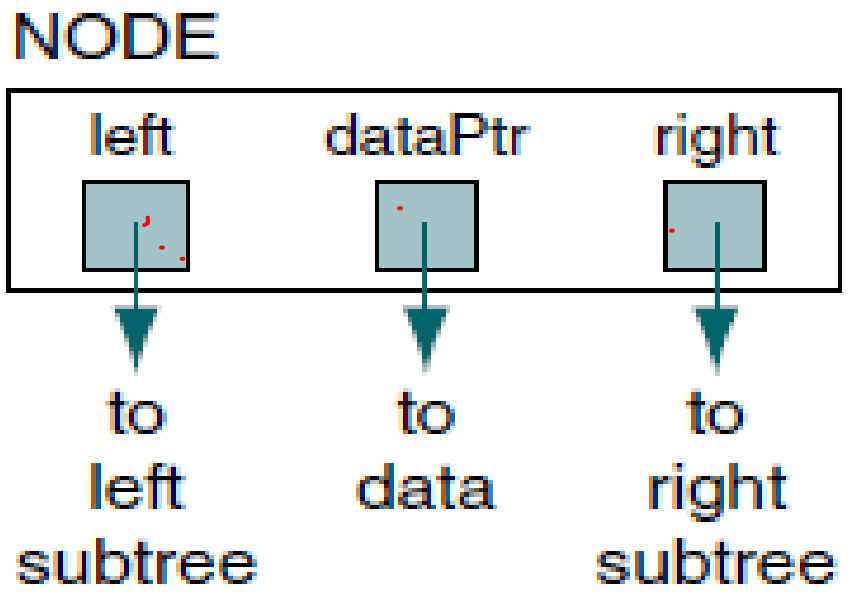
Binary Search Tree ADT.

Head: Count, a root pointer, and the address of the compare function needed to search the list.

Data Structure

Node: data pointer and two self-referential pointers to the left and right subtrees..

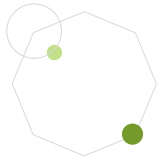




BST APPLICATIONS.

Integer Application

Student List Application



Integer Application:

Reading integers (values) from the keyboard and constructing BST

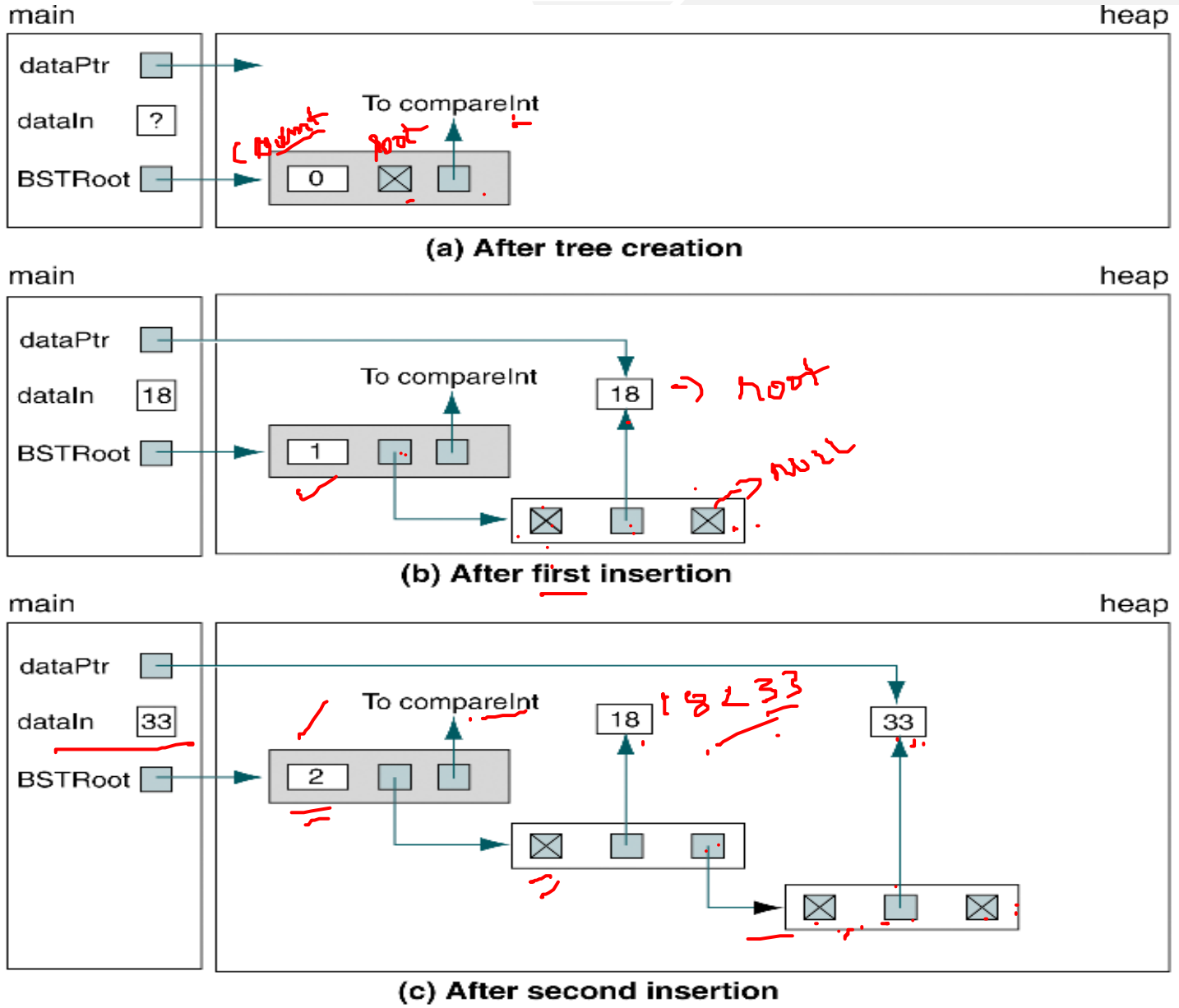
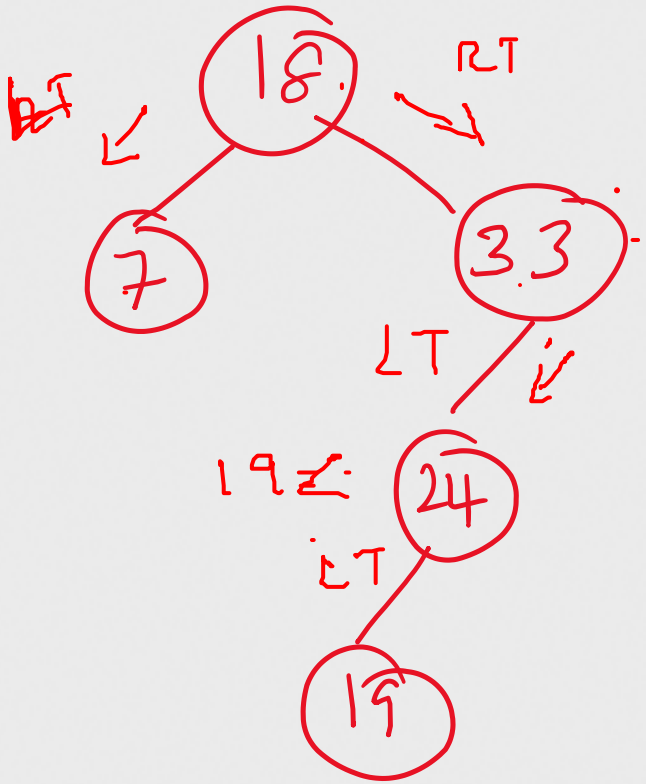


FIGURE 7-13 Insertions into a BST

PROGRAM 7-16

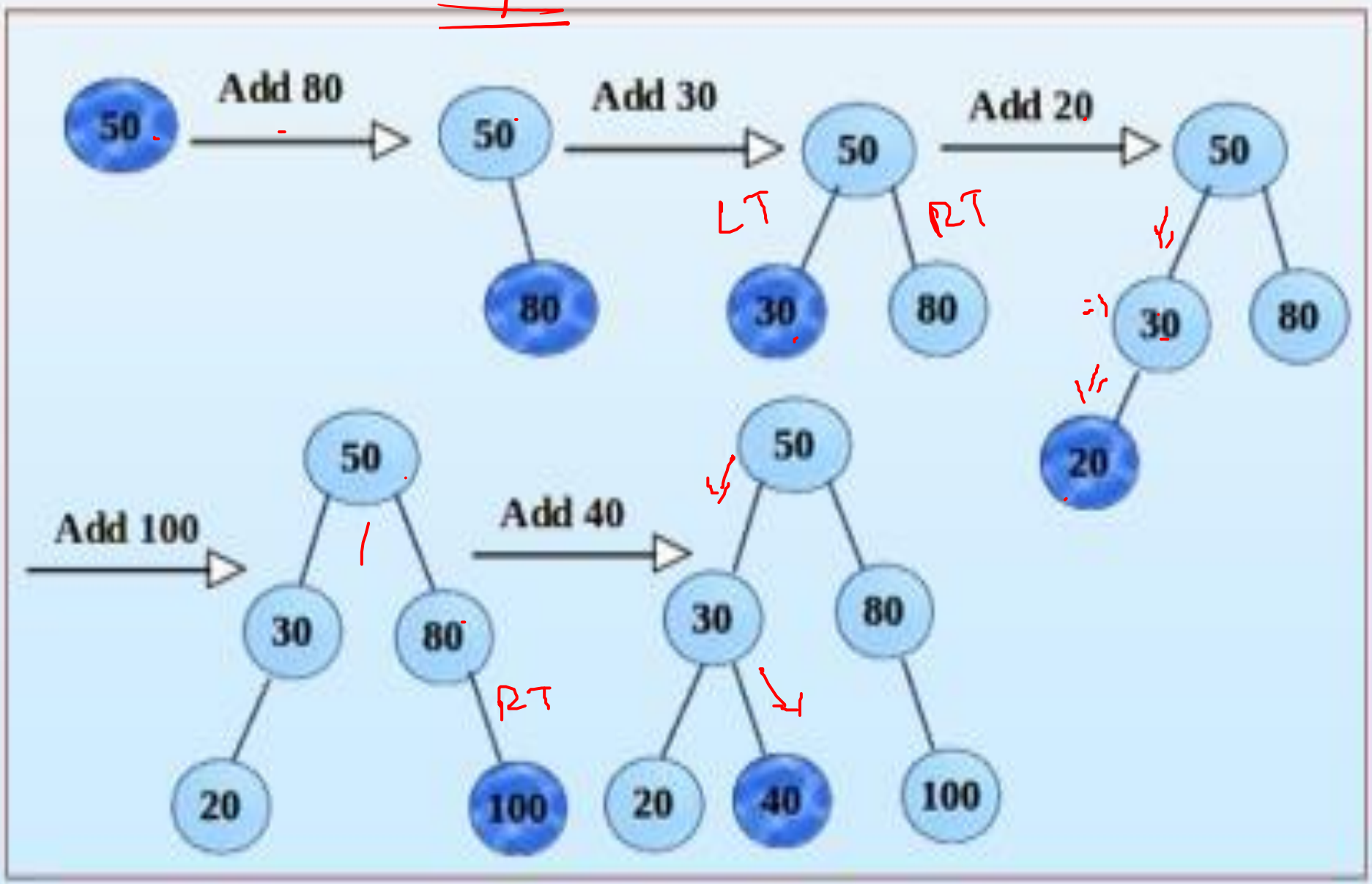
```
Results:
Begin BST Demonstation
Enter a list of positive integers;
Enter a negative number to stop.
Enter a number: 18
```

```
Enter a number: 33
Enter a number: 7
Enter a number: 24
Enter a number: 19
Enter a number: -1

BST contains:
   7
  18
  19
  24
  33

End BST Demonstration
```



example 2

50 ✓
80 ✓
30 ✓
20 ✓
100 ✓
40 ✓



STUDENT APPLICATION.

Three pieces of data: Student's ID, the student's name, and the student's grade-point average.

- Students are added and deleted from the keyboard.
- They can be retrieved individually or as a list

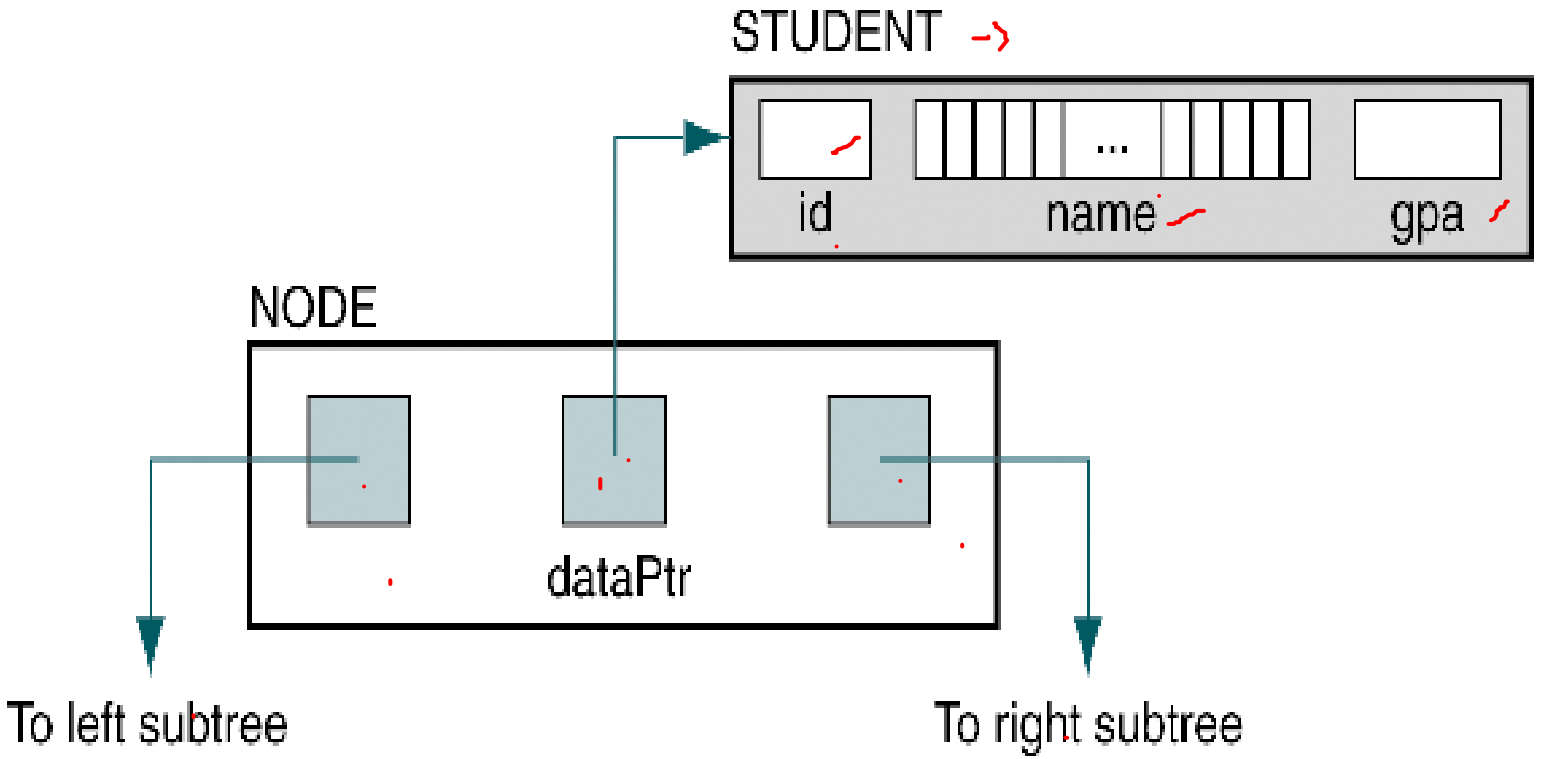


FIGURE 7-14 Student Data in ADT

Thank You.