ATME College of Engineering

Module 3

Additional list operations

Initially we have seen the different operations on the list like:

- Node Creation
- Inserting At the Beginning of the list
- Inserting At End of the list
- Deleting from the Beginning of the list
- Deleting from the End of the list
- Traversing/Display

Now lets have a look on additional list operations:

Other Operations:

- Find the length of the list
- search for a given key item
- creating an ordered linked list
- inserting a node whose position is given
- deleting a node whose position is given
- deleting a node whose information is given
- concatenating two lists
- reversing a list

1. Find the length of the list

```
cur = first;
while (cur != NULL)                                    /* As long as no end of list */
{
printf("%d\n", cur->info);                    /* Display the info of a node */
cur = cur->link;                                       /* point cur to the next node */
}
```

The above code displays the info field of each node in the list.

But, we want to find the number of nodes in the list.

This can be achieved very easily by replacing printf statement by count++ and initialize count to 0 before the loop.

```
int Length(NODE first)
{
NODE cur;
int count = 0;
if ( first == NULL )
 return 0;                                   /* Check for empty list. If so, return 0 */
cur = first;
while (cur != NULL)                          /* As long as no end of list */
{
count++;                                                  /* Update count by 1 */
cur = cur->link;                             /* point cur to the next node */
}
return count; }                              /* Return the length of list
}
```

2. Search for an item in a list

```
void search(int key, NODE first)
{
NODE cur;
if ( first == NULL )                                    /* check for empty list */
{
printf("List is empty\n");
return;
}
cur = first;
while (cur != NULL)                                     // As long as no end of list
{
if (key == cur->info) break;                            // If found go out of the loop
cur = cur->link;                                        // point cur to the next node
```

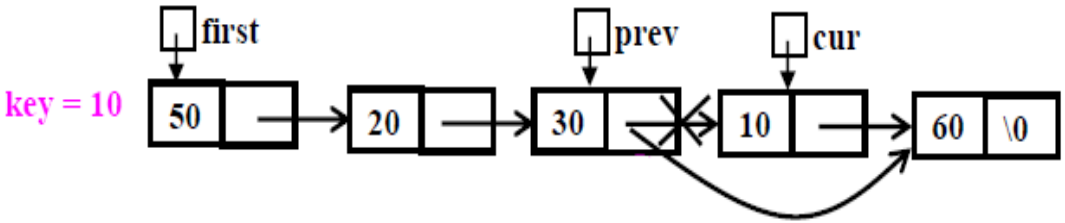
```
if (cur == NULL)                                    // If end of list, key not found
{
printf("Search is unsuccessful\n");
return;
}
printf("Serach is successful\n");                   // Yes, key found
}
```

3. Delete a node whose information field is specified

```
NODE delete_info(int key, NODE first)
{
NODE prev, cur;
if ( first == NULL )                                    // Check for empty list
{
printf("List is empty\n");
return NULL;
}
if ( key == first->info )                               // If key is present in first node, delete that node
{
cur = first;                                            // Save the address of the first node
first = first->link;                                    // Point first to second node in the list
free(cur);                                              // Delete the first node
return first; }                                         //Return second node as the first node
```

```
/* Search for the node to be deleted */
prev = NULL;
cur = first;
while (cur != NULL)                          /* As long as no end of list */
{
if (key == cur->info) break;                 /* If found go out of the loop */
prev = cur;                                  /* Save the address of cur node
cur = cur->link;                             /* point cur to the next node */
}
if (cur == NULL)                                    /* If end of list, key not found
{
printf("Search is unsuccessful\n");
return first;
}
```

```
/* Search successful. So, delete the node */
prev->link = cur->link;                    /* Establish link between the
                                              predecessor and successor
free(cur);                                 /* Delete the node with info key */
return first;                                  /* Return address of first node */
```



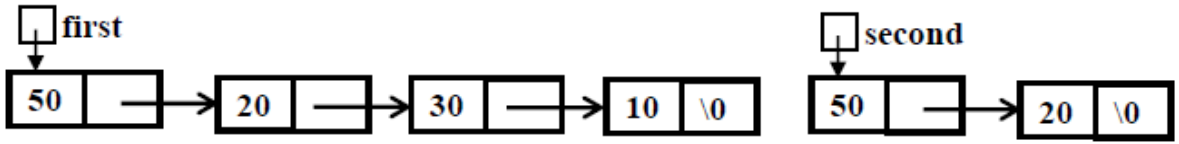
4. Concatenate two lists

- Concatenation of two lists is nothing but joining the second list at the end
- of the first list.
- Concatenation of two lists is possible if two lists exist. If one of the lists is empty, return the address of the first node of the non-empty list as shown below:

if (first == NULL) return second;

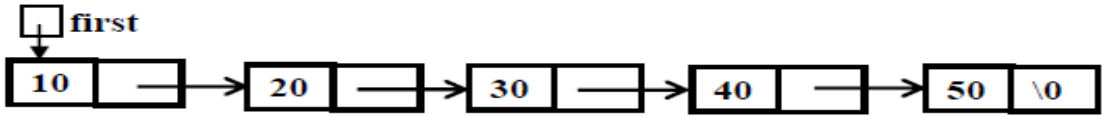
if (sec == NULL) return first;

Once, control comes out of second if-statement it means both lists are existing.

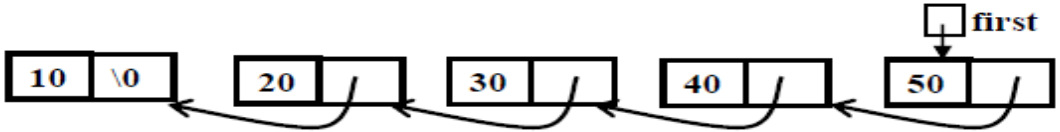


```
NODE concat (NODE first, NODE sec)
{
NODE cur;                                   /* holds the address of the last node of first list
if (first == NULL) return second;
if (sec == NULL) return first;
/* Obtain address of the last node of first list*/
cur = first;
while ( cur->link != NULL)
cur = cur->link;
cur->link = sec;                            /* Attach first node of second list to end of first list
return first;                               /* Return the first node of the concatenated list
}
```

5. Reverse (Invert) a list without creating new nodes



After reversing the list. the list is shown below:

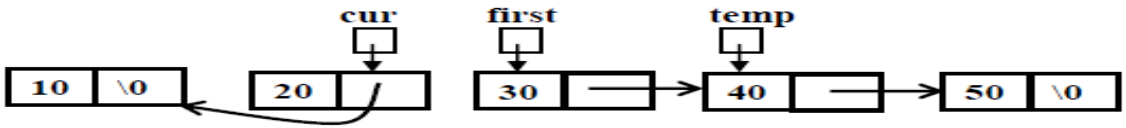


let us assume two things:

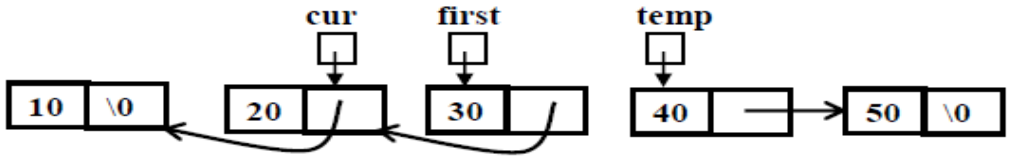
- The pointer variable cur always contains address of the first node of the partially reversed list (see above figure)
- The pointer variable first always contains address of the first node of the list to be reversed (see above figure)

Step 1: Obtain the address of the second node of the list to be reversed. This can be achieved using:

temp = first->link;



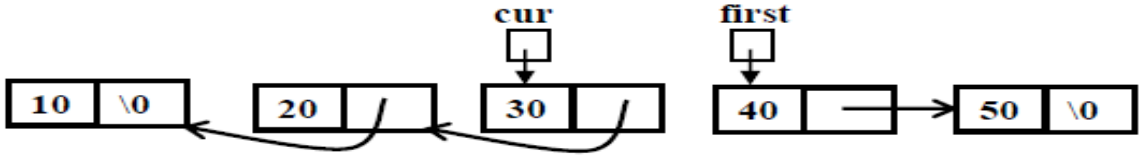
Step 2: Attach the first node of the list to be reversed, to the front end of the partially reversed list. This can be achieved using:



Step 3: The pointer variable cur always should contain address of the first node of the reversed list and first should contain address of first node of the list to be reversed.

cur = first;

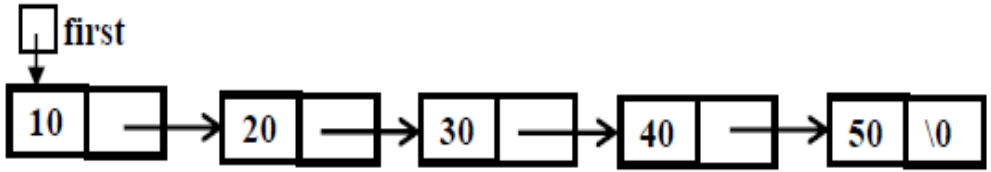
first = temp;



```
NODE reverse (NODE first)
{
NODE cur, temp;
cur = NULL;                                          /* Initial reversed list */
while (first != NULL)
{
temp = first->link;                                  // Obtain the address of second node
                                                     // of list to be reversed
first->link = cur;                                   // attach first node of the list to be reversed
                                                     // at the beginning of the partially reversed list
cur = first;                                         // Point cur to point to newly partially reversed list
first = temp;                                        // Point first to point to the list to be reversed
}
return cur;                                          /* Contains address of the reversed list */
```

6. Creating an ordered linked list

- A linked list in which the items are stored in some specific order is an ordered linked list.
- The elements in an ordered linked list can be in ascending or descending order or based on key information.
- A key is one or more fields within a structure that are used to identify the data

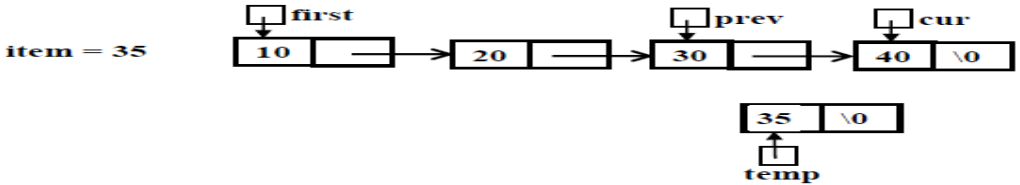


Step 1: Create a node to be inserted and insert the item

Step 2: If the node temp is inserted into the list for the first time (i.e., into the empty list), then return temp itself as the first node

Step 3: Inserting an item at the front end of the list: This case occurs, when the node to be inserted is less than the first node of the list.

Step 4: Inserting somewhere in the middle of the list



Step 5: Insert at middle/end.

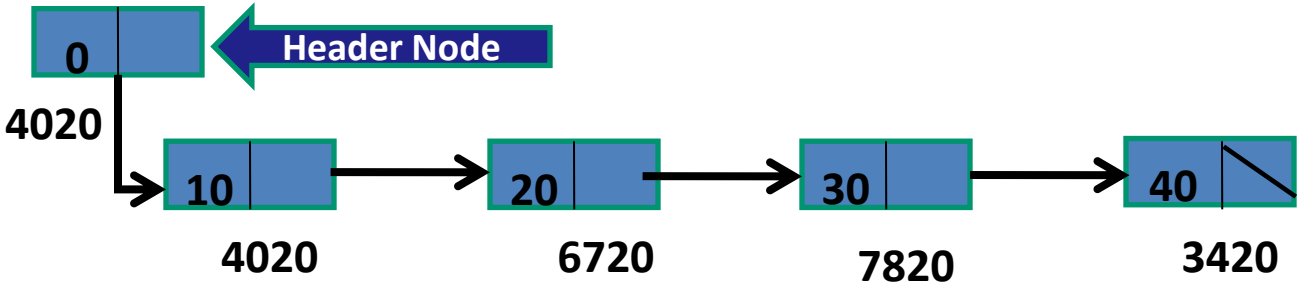
Step 6: Always we return the address of the first node

```
NODE insert(int item, NODE first)
{
NODE temp, prev, cur;
temp = getnode();                          /* obtain a node to be inserted */ STEP 1
temp->info = item;
temp->link = NULL;
/* Inserting the node for the first time                                    */ STEP 2
if ( first == NULL ) return temp;
/* Inserting the node in the beginning of the list                          */ STEP 3
if ( item <= first->info )
{
temp->link = first;
return temp;
```

```
/* find prev and cur locations so that node temp has to be inserted */
prev = NULL;                                                    /*STEP 4
cur = first;
while ( cur != NULL && item > cur->info )
{
prev = cur;
cur = cur->link;
}
/* Insert the node between prev and cur                          */ STEP 5
prev->link = temp;
temp->link = cur;
return first; /                                  * return the first node */ STEP 6
}
```

Header Node

- A header node in a linked list is a special node whose link field always contains address of the first node of the list.
- Using header node, any node in the list can be accessed.
- The info field of header node usually does not contain any information and such a node does not represent an item in the list.
- Sometimes, useful information such as, number of nodes in the list can be stored in the info field.
- If the list is empty, then link field of header node contains \0 (null).
- It is also called List Header



The advantages of a header node are shown below:

- Simplifies Insertion and deletion operations
- Designing of program will be very simple.
- Circular lists with header node are frequently used instead of ordinary linked lists because many operations can be easily implemented.

Header Node

```
void main( )
{    NODE head;
     head = getnode();
            head->info = 0;
            head->link = NULL;
           :
           :
    head =insert_front(item,head);
}
```

Insert @ Front

```
NODE insert_front(int item, NODE head)
{
NODE temp;
temp = getnode();                          /* Create a node, insert the item */
temp->info = item;

first = head->link;                        /* Obtain the address of the first node */
head->link = temp;                                  /* Insert at the front end */
temp->link = first;
return head;
}                                                   /* Return the header node */
```

Insert @ rear

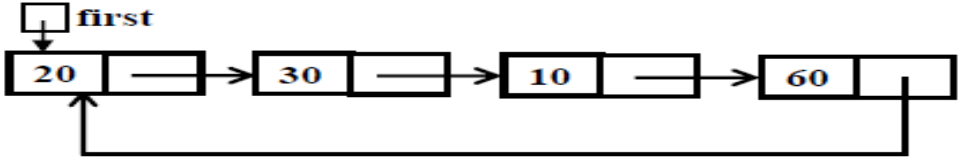
```
NODE insert_rear(int item, NODE head)
{
NODE temp, cur;
temp = getnode();                          /* node to be inserted */
temp->info = item;
temp->link = NULL;
cur = head->link;                          /* obtain address of the first node
while ( cur->link != NULL ) /              /* Obtain address of the last node
{
cur = cur->link;
}
cur->link = temp;                          /* insert node at the end
return head;  }
```

Circular list

ATME College of Engineering

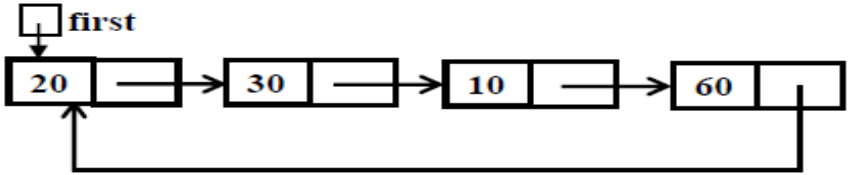
“What is circular list?”

Definition: A circular singly linked list is a singly linked list where the link field of last node of the list contains address of the first node.

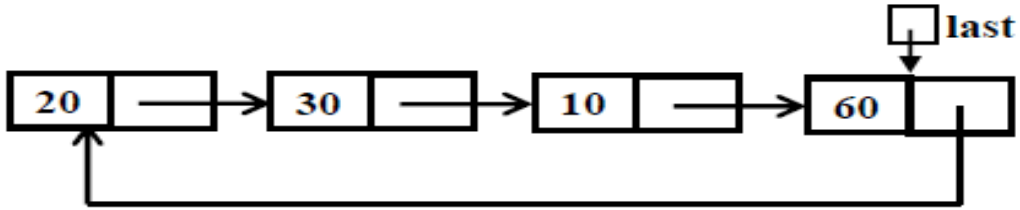


Since the list is circular, any node can be considered as first node and its predecessor is considered as last node.

Approach 1:



Approach 2 : pointer variable last can be used to designate the last node and the node that follow last, can be designated as the first.



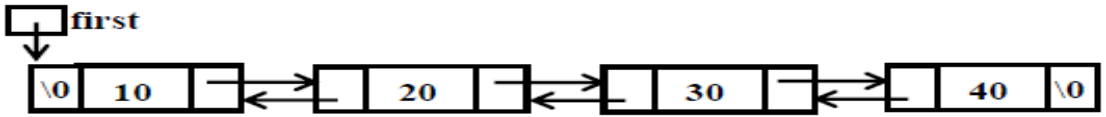
Some of the disadvantages of singly linked lists/circular lists:

- Using singly linked lists and circular lists it is not possible to traverse the list backwards. Two-way traversing is not possible.
- Insertion/Deletion to the left of a designated node x is difficult. This requires finding the predecessor of x which takes more time.

Doubly linked lists

A doubly-linked list is a linear collection of nodes where each node is divided into three parts:

- info – This is a field where the information has to be stored
- llink – This is a pointer field which contains address of the left node or previous node in the list
- rlink – This is a pointer field which contains address of the right node or next node in the list.



Using such lists, it is possible to traverse the list in forward and backward directions. Such a list where each node has two links is also called a two-way list.

Insert @ Front

```
NODE insert_front(int item, NODE first)
{
NODE temp;
temp = getnode();                          /*obtain a node from OS
temp->info = item;                         /* Insert an item into new node */
temp->llink = temp->rlink = NULL;
if (first == NULL) return temp;            /* Insert a node for the first time */
temp->rlink = first;                       /* Insert at the beginning of existing list */
first->llink = temp;
return temp;                               /* return address of new first node */
```

Insert @ rear

```
NODE insert_rear(int item, NODE first)
{
NODE temp, cur;
temp = getnode();                                   /*obtain a node from OS */
temp->info = item;                                  /* Insert an item into new node */
temp->llink = temp->rlink = NULL;
if (first == NULL)   return temp;                   /* Insert a node for the first time */
cur = first                                         /* Get the address of the first node */
while (cur->rlink != NULL)                          /* Find the address of the last node */
{
cur = cur->rlink;
}
cur->rlink = temp;                                  /* Insert the node at the end */
temp->llink = cur;
return first;                }                      /* return address of the first node */
```

Delete @ Front

```
NODE delete_front(NODE first)
{  NODE second;
if ( first == NULL )                                        /* Check for empty list */
{
printf("List is empty cannot delete\n");
return NULL;                                    /          / We can replace NULL with first also
}
if (first ->rlink == NULL)                          /* Delete if there is only one node */
{
        printf(“Item deleted = %d\n”, first->info);
        free(first);
        return NULL;
}
```

```
second = first->rlink;                    /* Get the address of second node */
second->llink = NULL;                     /* Make second node as the first node */
printf("Item deleted = %d\n", first->info);
free(first);                              /* Delete the first node */
return second;
```

Delete @ Rear

```
NODE delete_rear(NODE first)
{
NODE cur, prev;
if (first == NULL)                                   /* Check for empty list */
{
printf("List is empty cannot delete\n");
return first;
}
if ( first ->rlink == NULL )                         /*Only one node is present and delete it */
{
printf ("The item to deleted is %d\n",first->info);
free(first);                                         /* return to availability list */
return NULL; }                                       /* List is empty so return NULL */
```

```
/* Obtain address of the last node and just previous to that */
prev = NULL;
cur = first;
while( cur->rlink != NULL )
{
prev = cur;
cur = cur->rlink;
}
printf("The item deleted is %d\n", cur->info);
free(cur);                                                    /* delete the last node
*/
prev->rlink = NULL;                    /* Make last but one node as the last node */
return first;                                                 /* return address of the first node
```

Display DLL

```
void display(NODE first)
{
        NODE cur; int count = 0;
if ( first == NULL )                                    /* Check for empty list */
{
printf("List is empty\n");
return; }
printf("The contents of linked list\n");
cur = first;                                            /* Holds address of the first node */
while ( cur != NULL )                                   /* As long as no end of list */
{
printf("%d ",cur->info); count++;
cur = cur->rlink;                                       /* Point to the next node */
}
printf("Number of nodes = %d\n", count);}
```

Sparse matrix representation using multilinked data structure

A sparse matrix is a matrix that has very few non-zero elements spread out thinly. In other words, a matrix in which most of the elements are zeroes is called a sparse matrix.

	col0]	col[1]	col[2]
row[0]	10	20	30
row[1]	11	0	31
row [2]	12	22	32
row [3]	13	23	0

Not a sparse matrix

	col0]	col[1]	col[2]	col[3]
row[0]	10	0	0	40
row[1]	11	0	0	0
row [2]	0	0	0	0
row [3]	0	0	0	50
row [4]	0	15	0	25

Sparse matrix

- In the first matrix, 10 non-zero elements are present and 2 zero elements. So, it is not a sparse matrix.
- In the second matrix only 6 non-zero elements are present and 14 zero elements are present. So, it is a sparse matrix.

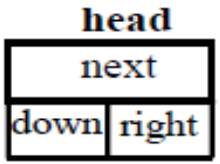
“What is the disadvantage of a sparse matrix?”

The sparse matrix contains many zeroes. If we are manipulating only non-zero values, then we are wasting the memory space by storing unnecessary zero values.

The above disadvantage can be overcome by storing only non-zero values thus saving the space.

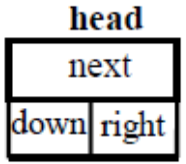
The sparse matrix can be more efficiently represented using linked list. In the linked representation:

Each column of a sparse matrix is represented as a circularly linked list with a header node having three fields: **down, right and next fields**



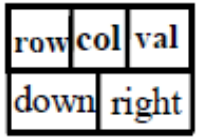
next – is the header of next column
right –not used
down – points to next item in the column

Each row of a sparse matrix is represented as a circularly linked with a header node having three fields: **down, right and next fields**



next – not used
right – points to next item in the row
down – not used

Each item is represented as a node having 5 fields as shown below

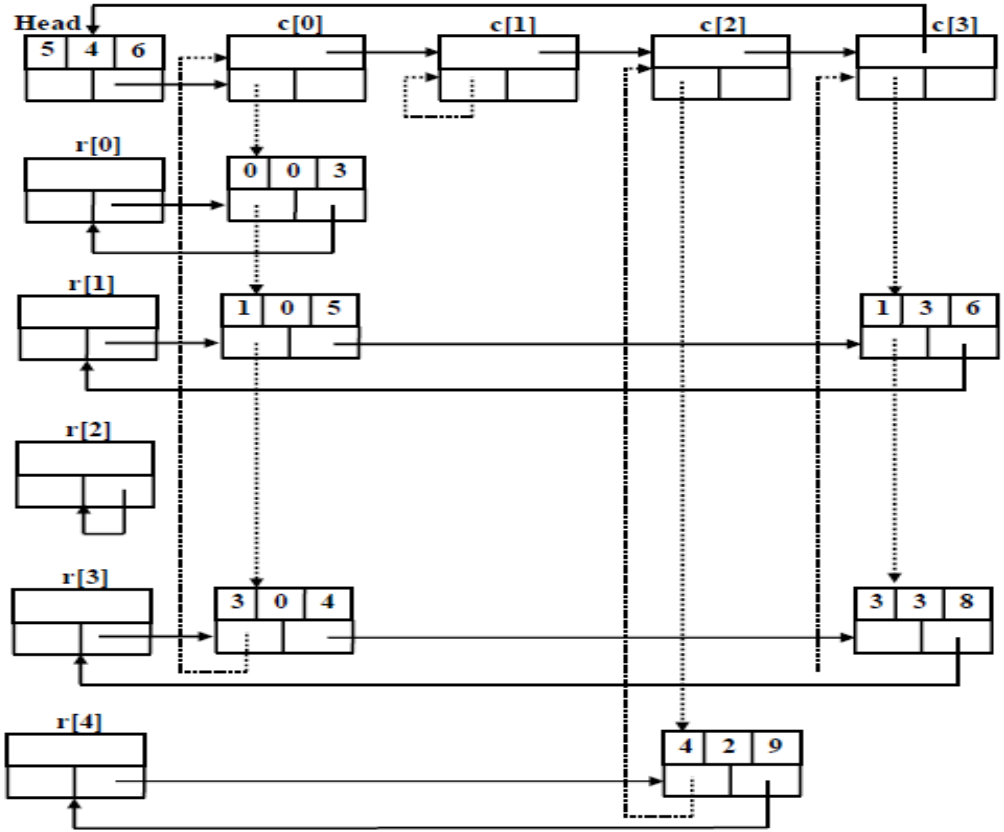


right – points to next item in the row
down – points to next item in the column

Now, consider the following 5 x 4 matrix:

	col[0]	col[1]	col[2]	col[3]
row [0]	3	0	0	0
row [1]	5	0	0	6
row [2]	0	0	0	0
row [3]	4	0	0	8
row [4]	0	0	9	0

The above matrix can be represented using linked list as



Note: The first node in the list identified by the variable head contains the size of the matrix where 5 represents the number of rows, 4 represent the number of columns and 6 represent non-zero elements to be manipulated.

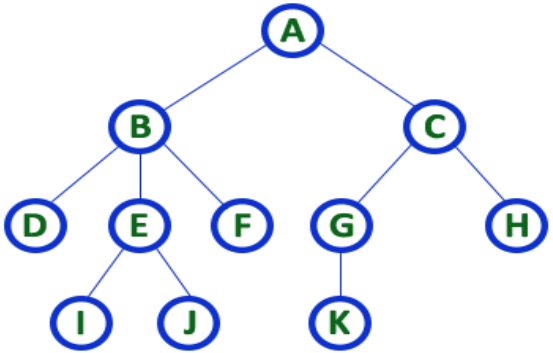
Tree Definition

Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

Tree is a set of finite set of one or more nodes that shows parent-child relation such that:

There is a special node called the root node

The remaining nodes are partitioned into disjoint subsets T1, T2, T3 ……Tn, $n \geq 0$ where T1, T2, T3 ……Tn which are all children of root node are themselves trees called subtrees.



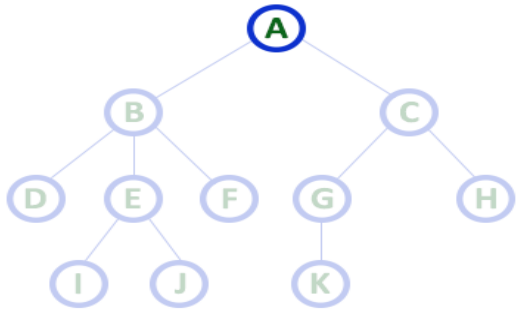
TREE with 11 nodes and 10 edges

- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

Tree terminology...

1.Root:

- The first node is called as Root Node.
- Every tree must have root node, there must be only one root node.
- Root n

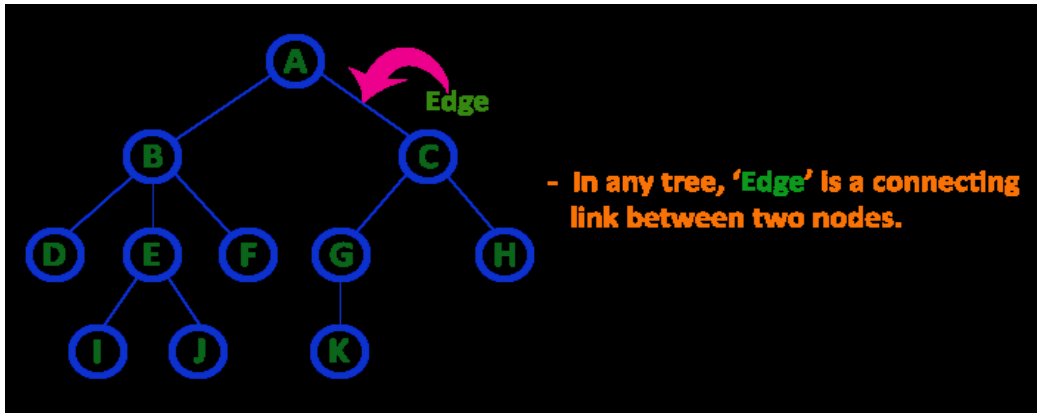


Here 'A' is the 'root' node

- In any tree the first node is called as ROOT node

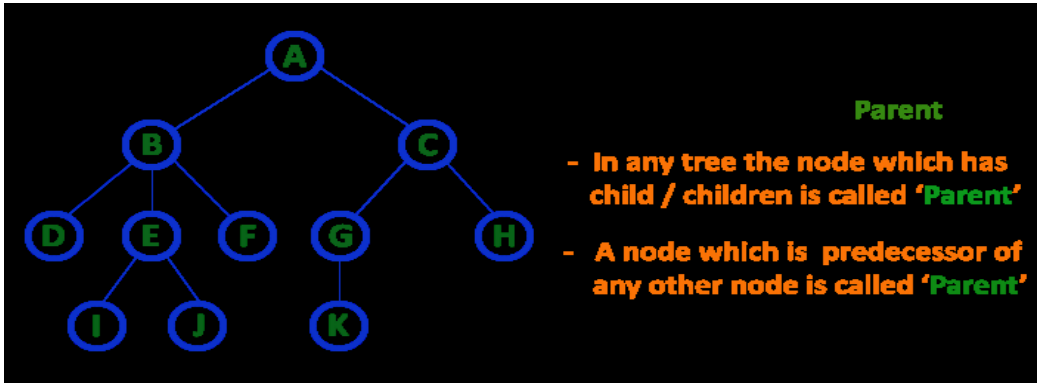
2. Edge

- In a tree data structure, the connecting link between any two nodes is called as EDGE.
- In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.



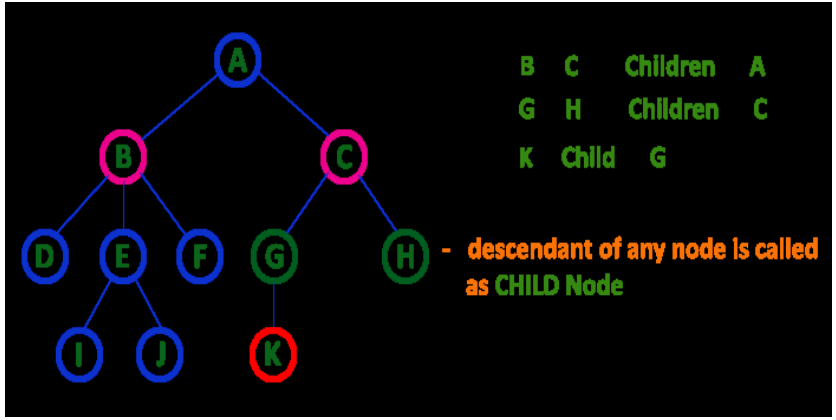
3. Parent

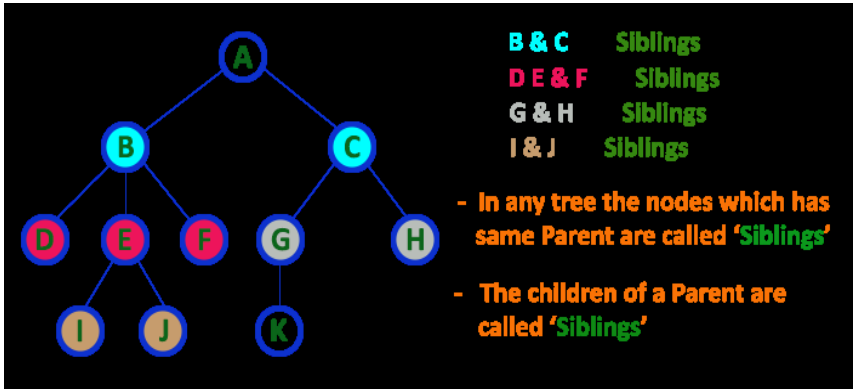
- In a tree data structure, the node which is predecessor of any node is called as PARENT NODE.
- In simple words, the node which has branch from it to any other node is called as parent node.
- Parent node can also be defined as "The node which has child / children".



4. Child

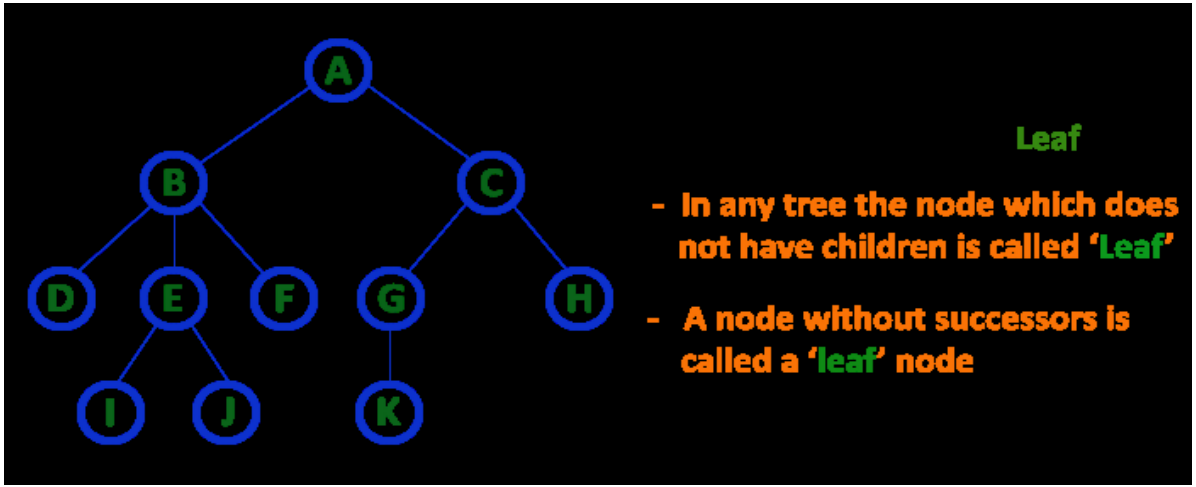
- The node which has a link from its parent node is called as child node.
- In a tree, any parent node can have any number of child nodes.
- In a tree, all the nodes except root are child nodes.
- The nodes with same parent are called as Sibling nodes.





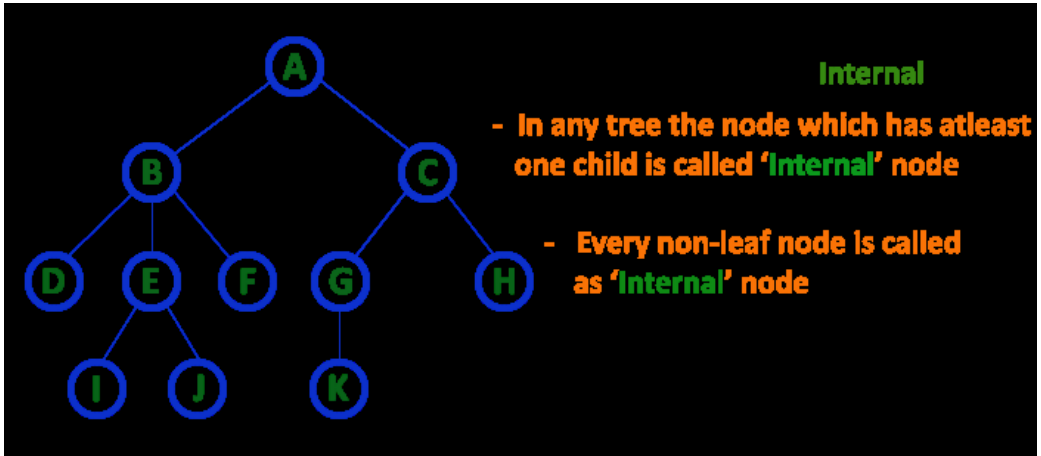
5. Leaf Node

- The node which does not have a child is called as LEAF Node.
- The leaf nodes are also called as External Nodes or 'Terminal' node.



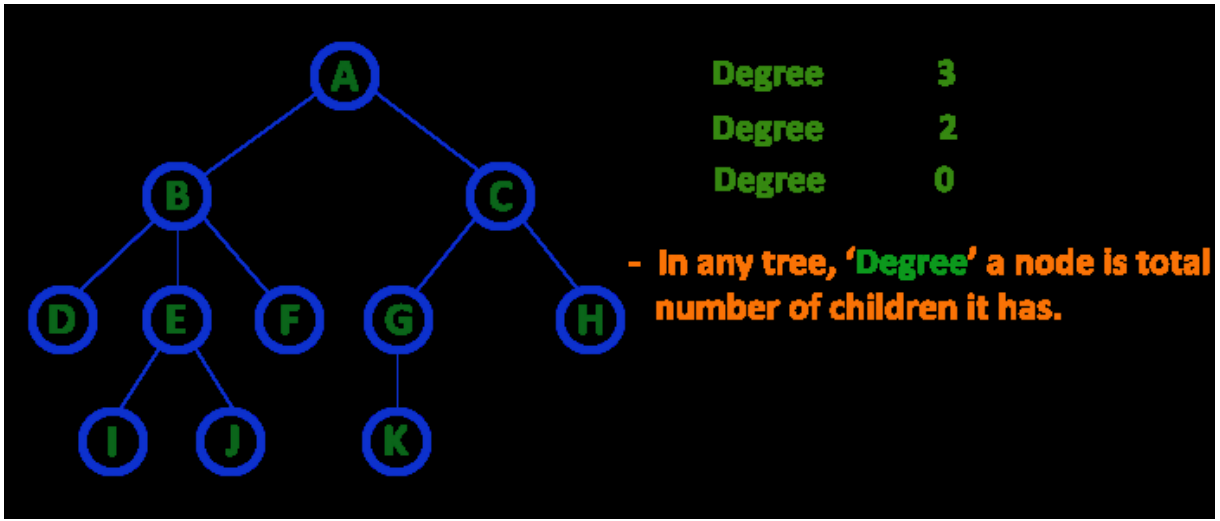
6. Internal Nodes

- An internal node is a node with atleast one child.
- Nodes other than leaf nodes are called as Internal Nodes.
- The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes.



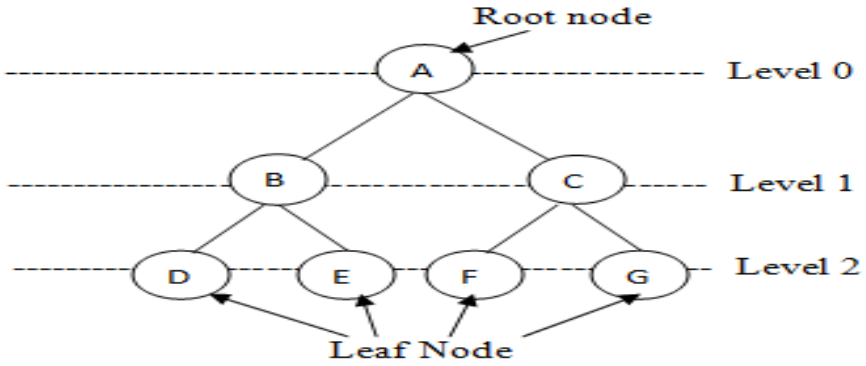
7. Degree

- In a tree data structure, the total number of children of a node is called as DEGREE of that Node.



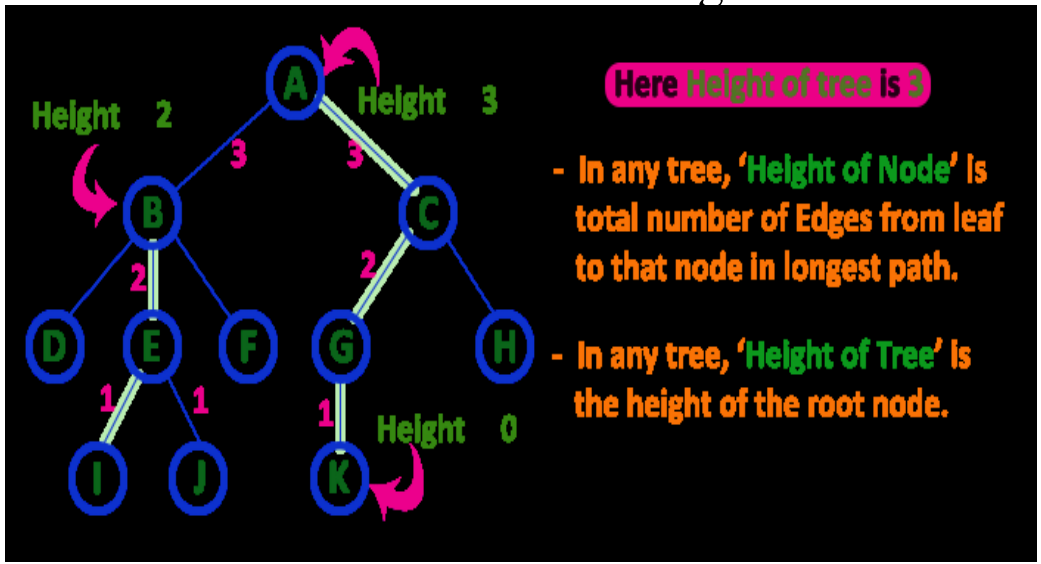
8. Level

- In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on...
- In a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



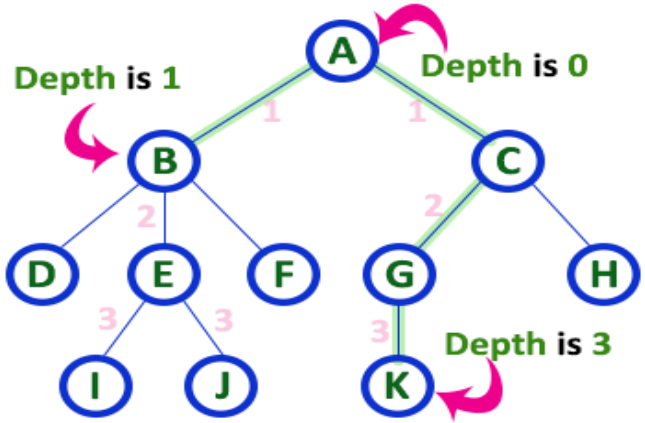
9. Height

- the total number of egdes from leaf node to a particular node in the longest path is called as HEIGHT of that Node.
- In a tree, height of the root node is said to be height of the tree.



10. Depth

- In a tree data structure, the total number of egdes from root node to a particular node is called as DEPTH of that Node.

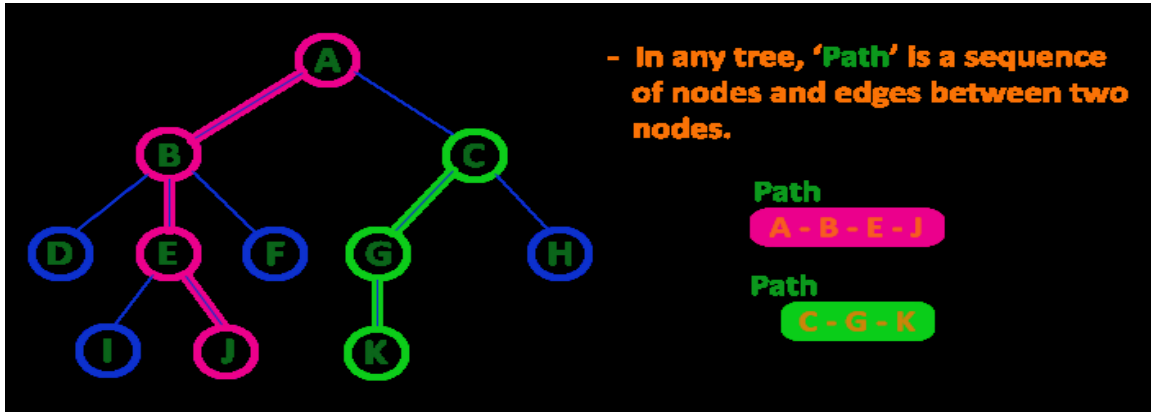


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

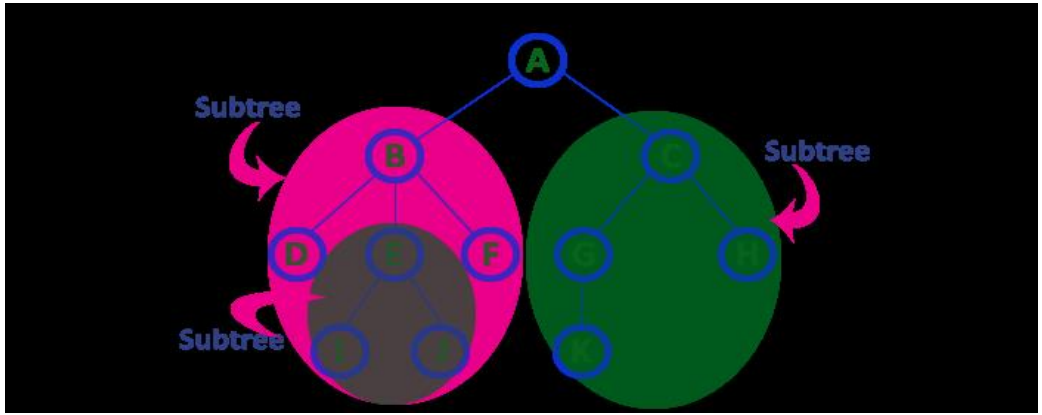
11. Path

- In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes.
- Length of a Path is total number of nodes in that path. In below example the path A - B - E - J has length 4.



12. Sub Tree

- Every child node will form a subtree on its parent node.

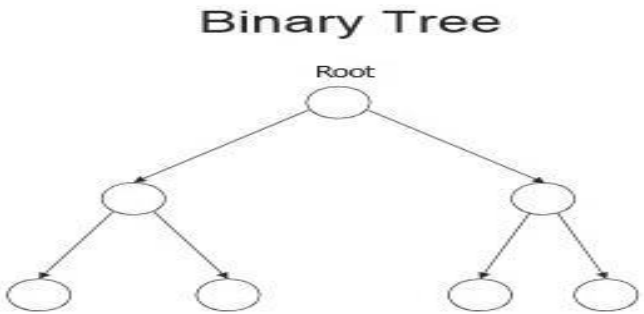


Type of Trees

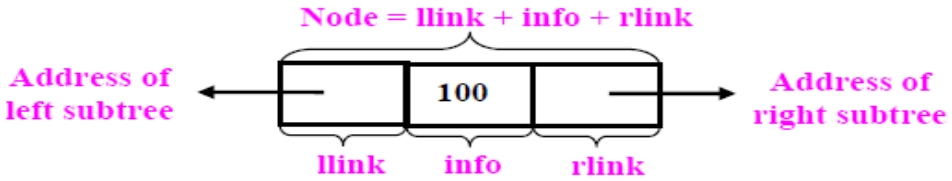
- General tree
- Binary tree
- Binary Search Tree

Binary tree

- A Binary tree is a data structure in that each node has at most **two nodes** left and right. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.
- In binary tree, root has **in-degree 0** and maximum **out-degree 2**.
- In binary tree, each node have in-degree **one** and maximum out-degree **2**.
- Height of a binary tree is : Height(T) = { **max** (Height(Left Child) , Height(Right Child) + 1}
-

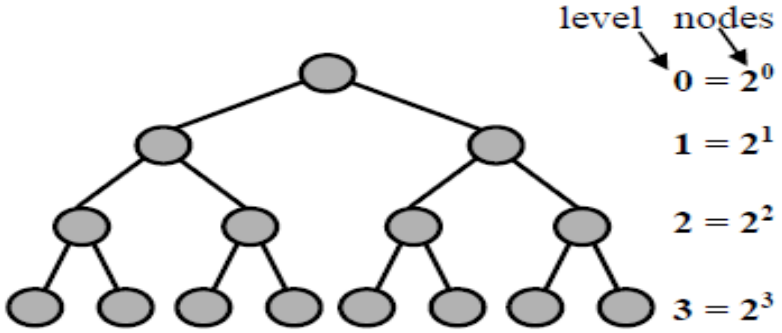


- A binary tree can be partitioned into three subgroups namely root, left subtree and right subtree.
- **Root** – If tree is not empty, the first node in the tree is called root node.
- **left subtree** – It is a tree which is connected to the left of root. Hence it is called left subtree.
- **right subtree** – It is a tree which is connected to the right of root. Hence it is called right subtree.
- The pictorial representation of a typical node in a binary tree is shown below:



Properties of binary trees

- a) The maximum number of nodes on level *i* of a binary tree = 2^i for $i \geq 0$
- b) The maximum number of nodes in a binary tree of depth *k* = $2^k - 1$
- Total number of nodes nt = $2^{i+1} - 1$ Substituting i = 3 we get nt = 15 which is total number of nodes in a fully binary tree
- The depth of the tree k = maximum level + 1 = i + 1
- Substituting this value ir



Representation of Binary Tree

1. Array Representation
2. Linked List Representation.

Representation of Binary Tree

```
Struct node
{

int data;
struct node *  left, *right;
};
```

Array Representation

1. To represent a tree in one dimensional array nodes are marked sequentially from left to right start with root node.
2. First array location can be used to store no of nodes in a tree.

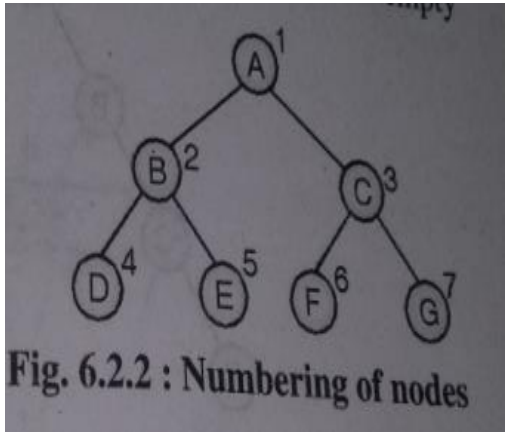
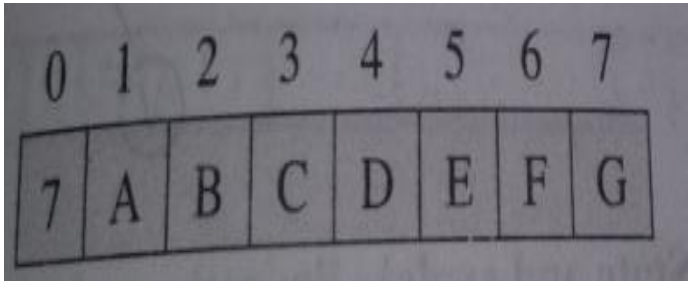


Fig. 6.2.2 : Numbering of nodes

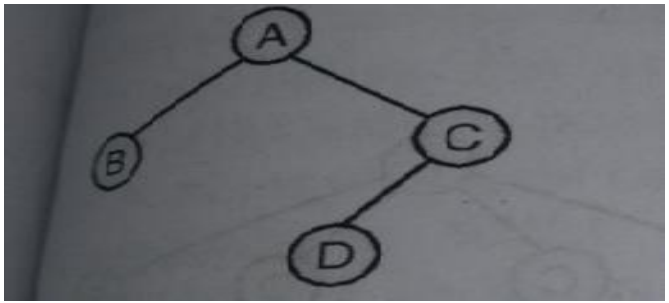


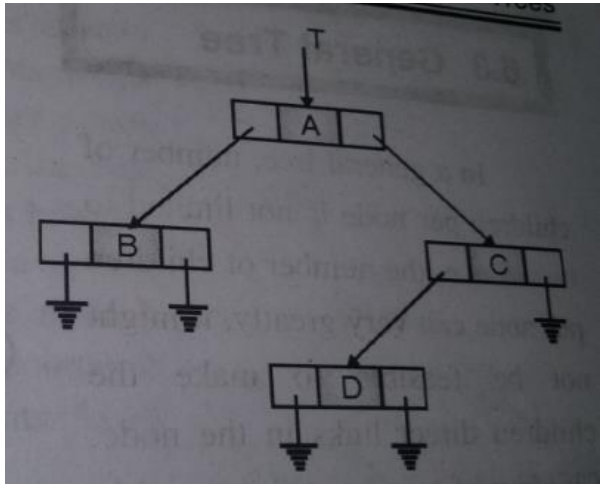
Linked Representation

This type of representation is more efficient as compared to array.

1. Left and right are pointer type fields left holds address of left child and right holds address of right child.

2.
```
Struct node
{
int data;
struct node * left,*right;
};
```





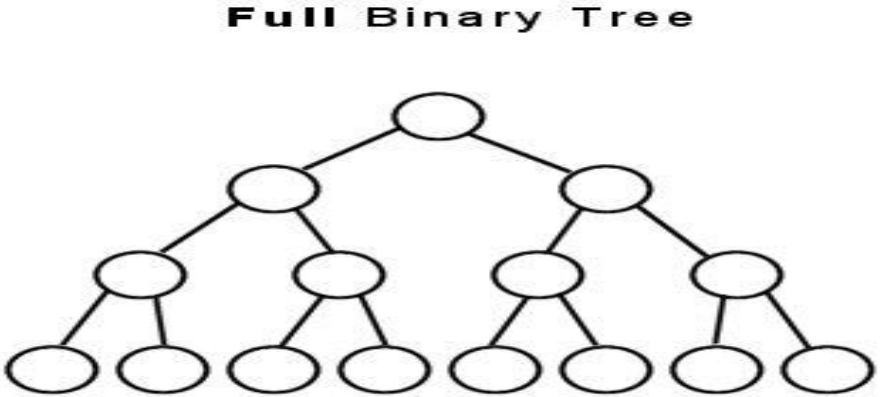
Binary Tree Types

1. Full Binary Tree
2. Extended Binary Tree
3. Complete Binary Tree
4. Skewed Binary Tree

Full Binary Tree

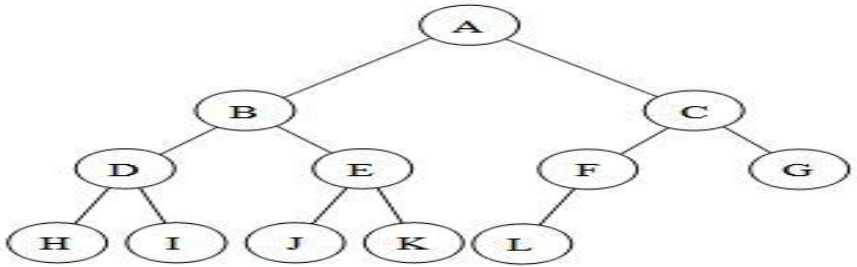
- A binary tree having 2i nodes in any given level i is called full binary tree.
- A **full binary tree** (sometimes proper **binary tree** or 2-**tree**) is a **tree** in which every node other than the leaves has two children or no childeren.
- Every level is completely filled up.

No of nodes= 2^{h+1} -1



Complete Binary Tree

A **complete binary tree** is a **tree** in which



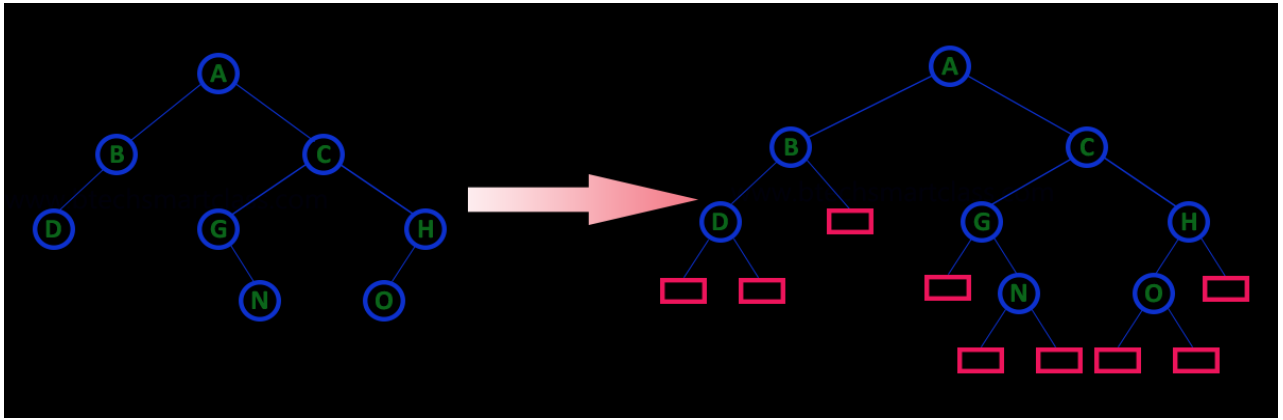
1. All leaf nodes are at n or n-1 level
1. All the levels are filled, except the last level.
2. Levels are filled from left to right
3. The maximum number of nodes in complete binary tree is 2^h
4. The minimum height of a complete binary tree is $\log_2(n+1) - 1$.

Extended Binary Tree

An **extended binary tree** is a transformation of any **binary tree** into a Complete **binary tree**.

This transformation consists of replacing every null subtree of the original **tree** with “special nodes”or “failure nodes” .

The nodes from the original **tree** are then internal nodes, while the “special nodes” are external nodes.



Skewed Binary Tree

- A binary tree is said to be ***Skewed Binary Tree or Degenerate Binary Tree*** if every node in the tree contains either only left or only right sub tree.
- If the node contains only left sub tree then it is called *left-skewed binar tree*
- if the tree contains only right sub tree then it is called *right-skewed binary tree.*

Binary Tree program in C

```
#include<stdio.h>
      struct node
     {
         int data;
         struct node *left, *right;
      }
      void main()
    {
       struct node *root;
       root = create();
    }
struct node *create()
{
struct node *temp;
  int data;
temp = (struct node *)malloc(sizeof(struct
node));
   printf("Press 0 to exit");
   printf("\nPress 1 for new node");
   printf("Enter your choice : ");
   scanf("%d", &choice);
   if(choice==0)
{
return 0;
}
```

```
else
{
   printf("Enter the data:");
   scanf("%d", &data);
   temp->data = data;
   printf("Enter the left child of %d", data);
   temp->left = create();
printf("Enter the right child of %d", data);
temp->right = create();
return temp;
}
}
```

Binary Tree Traversal

- Traversing is a method of visiting each node of a tree exactly once in a systematic order.
- During traversal, we may print the **info** field of each node visited.

Different traversal techniques of a binary tree

1. Preorder traversal:-In this traversal method first process root element, then left sub tree and then right sub tree.

Procedure:-

Step 1: Visit root node

Step 2: Visit left sub tree in preorder

Step 3: Visit right sub tree in preorder

Function to traverse the tree in preorder

```
void preorder(NODE root)
{
if ( root == NULL ) return;
printf("%d ",root->info);                    /* visit the node */
preorder(root->llink);                       /* visit left subtree in preorder*/
preorder(root->rlink);                       * visit right subtree in preorder*/
}
```

2. Inorder traversal:-

In this traversal method first process left element, then root element and then the right element.

Procedure:-

Step 1: Visit left sub tree in inorder

Step 2: Visit root node

Step 3: Visit right sub tree in inorder

Function to traverse the tree in inorder

```
void inorder(NODE root)
{
if ( root == NULL ) return;
inorder(root->llink);                /* visit left-subtree in inorder */
printf("%d ",root->info);                        /* visit the node */
inorder(root->rlink);               /* visit right-subtree in inorder */
}
```

3. Postorder traversal:-

In this traversal first visit / process left sub tree, then right sub tree and then the root element.

Procedure:-

Step 1: Visit left sub tree in postorder

Step 2: Visit right sub tree in postorder

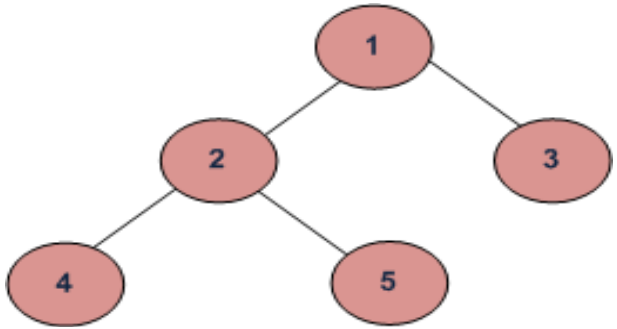
Step 3: Visit root node

Function to traverse the tree in postorder

```
void postorder(NODE root)
{
if ( root == NULL ) return;
postorder(root->llink);          /* visit leftsubtree in postorder*/
postorder(root->rlink);          /* visit rightsubtree in postorder*/
printf("%d ", root->info);       /* visit the node */
}
```

4. Level order traversal:

- The nodes in a tree are numbered starting with the root on level 0, continuing with the nodes on level 1, level 2 and so on.
- Nodes on any level are numbered from left to right.
- Visiting the nodes using the ordering suggested by the node numbering is called level ordering traversing.
- Firstly visit the root, then the root‟s left child, followed by the root‟s right child. Thus continuing in this manner, visiting the nodes at each new level from the leftmost node to the rightmost node.



- Level order traversal: 1 2 3 4 5
- Initially in the code for level order add the root to the queue.
- The function operates by deleting the node at the front of the queue, printing the nodes data field and adding the nodes left and right children to the queue.

```
void level_order(NODE root)
{
NODE q[MAX_QUEUE], cur;
int front = 0, rear = -1;                                   // queue is empty
q[++rear] = root;                                           // Insert root node into queue
while (front <= rear)                                       // As long as queue is not empty
{
cur = q[front++];                                           // Delete from queue
printf("%d ", cur->info);                                   // visit the node
if (cur->llink != NULL)                                     // Insert left subtree into queue
q[++rear] = cur->llink;
if (cur->rlink != NULL)                                     // Insert right subtree into queue
q[++rear] = cur->rlink;
}
printf("\n");
}
```

5. Iterative inorder traversal

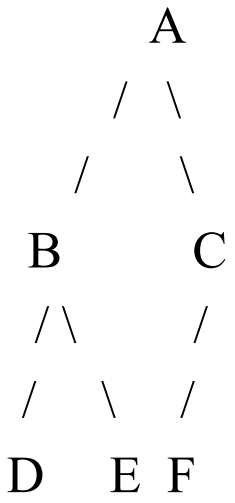
- Iterative inorder traversal explicitly make use of stack function.
- The left nodes are pushed into stack until a null node is reached, the node is then removed from the stack and displayed.
- The node‟s right child is stacked until a null node is reached.
- The traversal then continues with the left child.
- Thetraversal is complete when the stack is empty.

```
void inorder(NODE root)
{
NODE cur,s[20];
int top = -1;
if ( root == NULL )
{
printf("Tree is empty\n");
return;
}
cur = root;
for (;;)
{
while ( cur != NULL )
{
s[++top] = cur;          /* push(cur,&top,s); */
cur = cur->llink;        /* traverse left*/
}
if (top != -1 )          /* If stack not empty */
{
cur = s[top--];          /* Remove recent node from stack */
printf("%d ",cur->info);  /* visit node */
cur = cur->rlink;        /  * traverse right */
}
else
return;   }}
```

Construct Tree from given Inorder and Preorder traversals

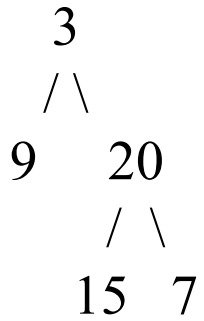
Inorder sequence: D B E A F C

Preorder sequence: A B D E C F



preorder = 3,9,20,15,7

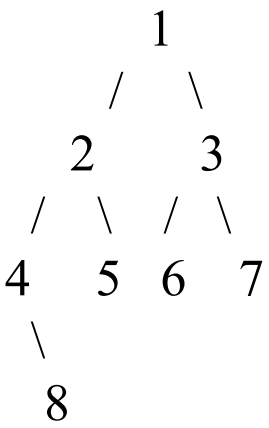
inorder = 9,3,15,20,7



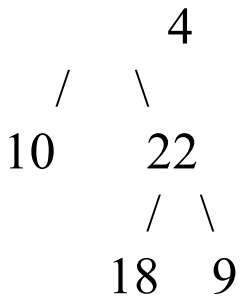
Construct Tree from given Inorder and Postorder traversals

inorder = 4, 8, 2, 5, 1, 6, 3, 7
postorder = 8, 4, 5, 2, 6, 7, 3, 1

Output: Root of below tree

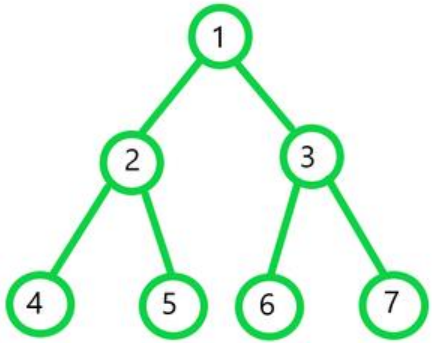


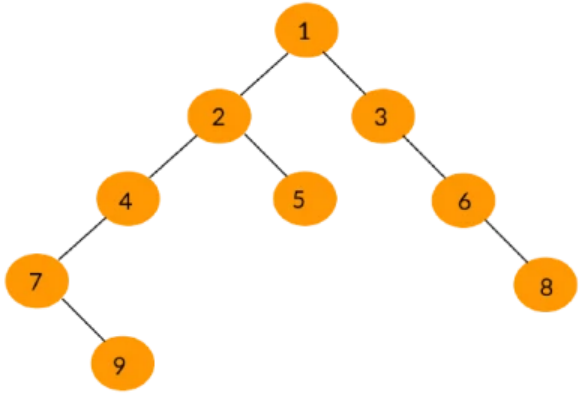
Postorder = 10, 18, 9, 22, 4
Inorder = 10, 4, 18, 22, 9



Construct Tree from given Preorder and Postorder traversals

Consider the below given binary tree. write the inorder, preorder and postorder traversal values

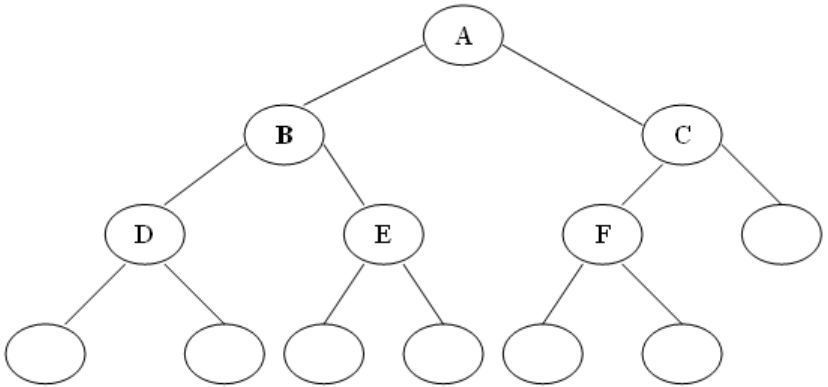




Threaded Binary Tree

Threaded Binary Tree

- In a linked representation of a binary tree, the number of null links (null pointers) are actually more than non-null pointers.
- Consider the following binary tree:



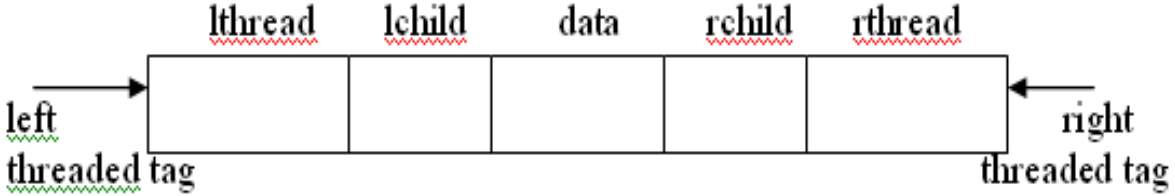
A Binary tree with the null pointers

- In above binary tree, there are 7 null pointers & actual 5 pointers.
- In all there are 12 pointers.
- We can generalize it that for any binary tree with n nodes there will be (n+1) null pointers and 2n total pointers.
- The objective here to make effective use of these null pointers.
- A. J. perils & C. Thornton jointly proposed idea to make effective use of these null pointers.
- According to this idea we are going to replace all the null pointers by the appropriate pointer values called threads.

- And binary tree with such pointers are called threaded tree.
- In the memory representation of a threaded binary tree, it is necessary to distinguish between a normal pointer and a thread.

Threaded Binary Tree

- Therefore we have an alternate node representation for a threaded binary tree which contains five fields as show bellow:



For any node p, in a threaded binary tree.

lthred(p)=1 indicates lchild (p) is a thread pointer
lthred(p)=0 indicates lchild (p) is a normal
rthred(p)=1 indicates rchild (p) is a thread
rthred(p)=0 indicates rchild (p) is a normal pointer

- Also one may choose a one-way threading or a two-way threading.
- Here, our threading will correspond to the in order traversal of T.

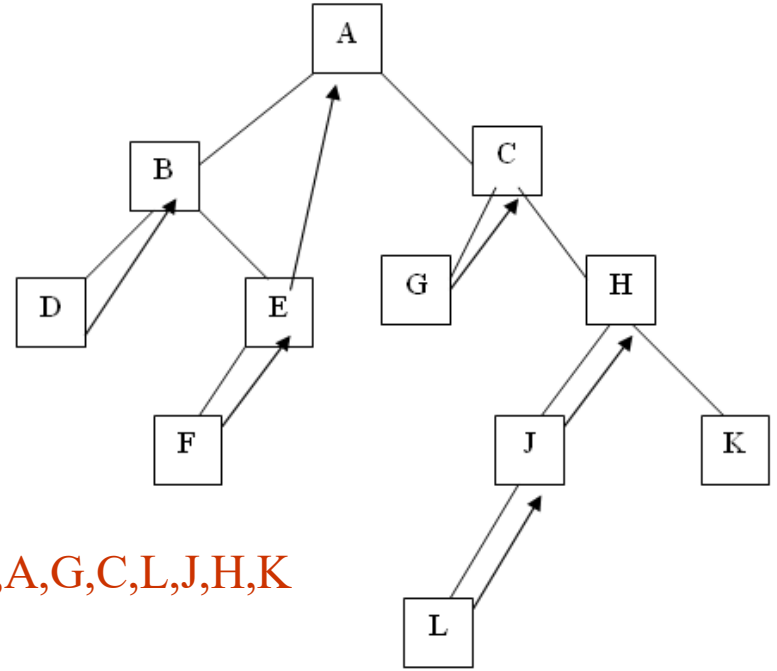
Threaded Binary Tree

One-Way

- Accordingly, in the one way threading of T, a thread will appear in the right field of a node and will point to the next node in the **in-order** traversal of T.
- See the bellow example of one-way **in-order** threading.

Threaded Binary Tree

One-Way



Inorder of bellow tree is: D,B,F,E,A,G,C,L,J,H,K

One-way inorder threading

Threaded Binary Tree
Two-Way

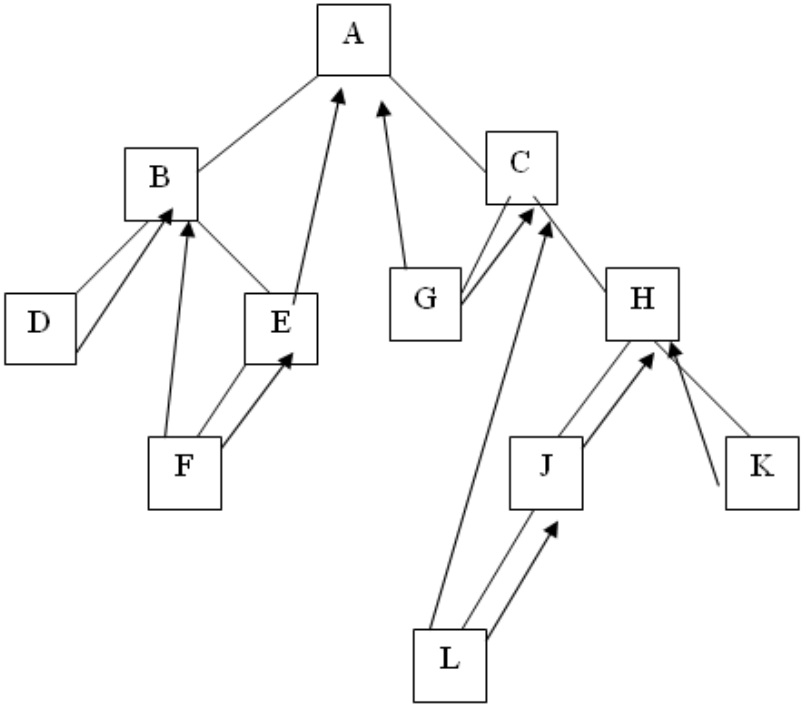
- In the two-way threading of T.
- A thread will also appear in the left field of a node and will point to the preceding node in the **in-order** traversal of tree T.
- Furthermore, the left pointer of the first node and the right pointer of the last node (in the **in-order** traversal of T) will contain the null value when T does not have a header node.

Threaded Binary Tree

- Bellow figure show two-way **in-order** threading.
- Here, right pointer=next node of **in-order** traversal and left pointer=previous node of **in-order** traversal
- Inorder of below tree is: D,B,F,E,A,G,C,L,J,H,K

Threaded Binary Tree

Inorder of bellow tree is:
D,B,F,E,A,G,C,L,J,H,K



Two-way inorder threading

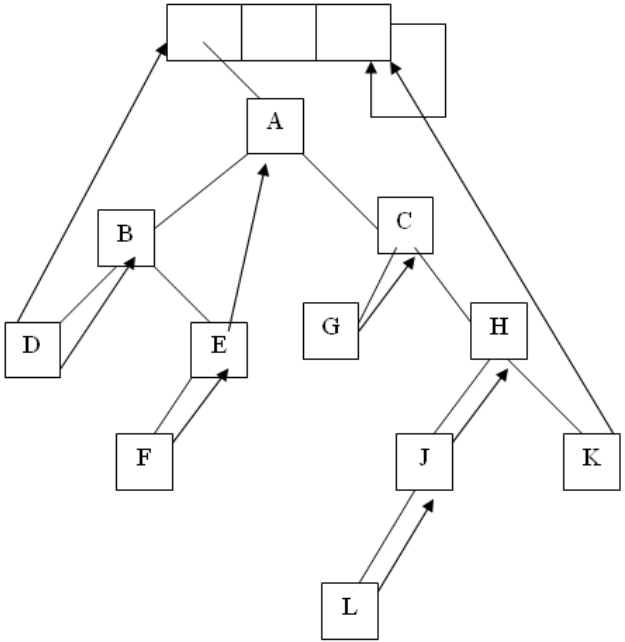
Threaded Binary Tree

Two-way Threading with Header node

- Again two-way threading has left pointer of the first node and right pointer of the last node (in the inorder traversal of T) will contain the null value when T will point to the header nodes is called two-way threading with header node threaded binary tree.

Threaded Binary Tree

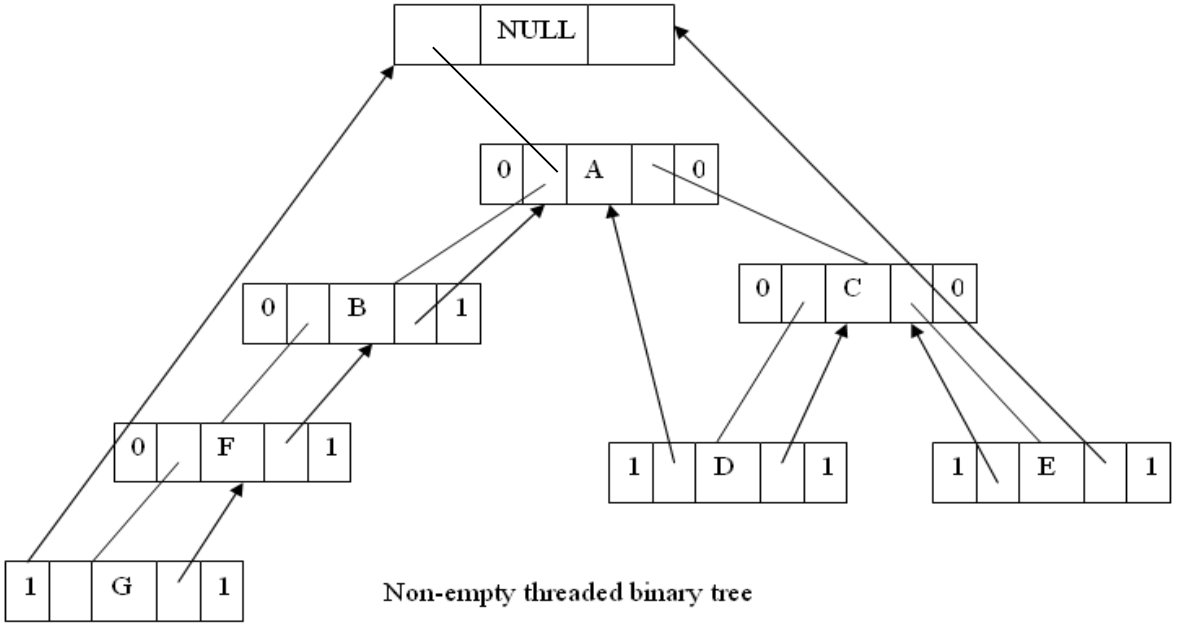
- Bellow figure to explain two-way threading with header node.



Threaded Binary Tree

- Below example of link representation of threading binary tree.
- **In-order** traversal of below tree: G,F,B,A,D,C,E

Threaded Binary Tree



Non-empty threaded binary tree

Advantages of threaded binary tree:

Threaded binary trees have numerous advantages over non-threaded binary trees listed as below:

–The traversal operation is more faster than that of its unthreaded version, because with threaded binary tree non-recursive implementation is possible which can run faster and does not require the botheration of stack management.

–Can efficiently determine the predecessor and successor nodes starting from any node. In case of unthreaded binary tree, however, this task is more time consuming and difficult. For this case a stack is required to provide upward pointing information in the tree

- Any node can be accessible from any other node.
- Threads are usually more to upward whereas links are downward. Thus in a threaded tree, one can move in their direction and nodes are in fact circularly linked. This is not possible in unthreaded counter part because there we can move only in downward direction starting from root.

Disadvantages of threaded binary tree:

- Insertion and deletion from a threaded tree are very time consuming operation compare to non-threaded binary tree.
- This tree require additional bit to identify the threaded link.

Thus a node can be defined as

```
struct node
{
int info;
struct node *llink;                    /* Pointer to the left subtree */
struct node *rlink;                    /* Pointer to the right subtree */
int lthread;                           /* 1 indicates a thread else ordinary link
*/
int rthread;
};
typedef struct node* NODE
```

Inorder Traversal of a Threaded Binary Tree

- An inorder traversal can be performed without making use of a stack.
- For any node, ptr, in a threaded binary tree, if ptr→rightThread =TRUE, the inorder successor of ptr is ptr →rightChild by definition of the threads.
- Otherwise we obtain the inorder successor of ptr by following a path of left-child links from the right-child of ptr until we reach a node with leftThread = TRUE.
- The function insucc ( ) finds the inorder successor of anynode in a threaded tree without using a stack.

Inorder traversal of right in-thread

```
void inorder(NODE head)
{
NODE temp;
if ( head->llink == head )
{
printf("Tree is empty\n");
return;
}
printf("The inorder traversal of the tree is\n");
temp = head;
for (;;)
{
temp = inorder_successor(temp);
if ( temp == head ) return;
printf("%d ",temp->info);
}
}
```

Threaded binary trees

"What are the disadvantages of binary trees?"

- In a binary tree, more than 50% of link fields have \0 (null) values and more memory space is wasted by storing \0 (null) values
- Traversing a tree with binary tree is time consuming. This is because, the traversal of a tree uses stack.
- Most of the time is spent in pushing and popping activities during traversing.
- Computations of predecessor and successor of given nodes is time consuming
- In binary trees, only downward movements are possible

All these disadvantages can be overcome using **threaded binary tree.**

Defining Threaded Binary Trees

- In a binary tree, there are many nodes that have an empty left child or empty right child or both.
- We can utilize these fields in such a way so that the empty left child of a node points to its inorder predecessor and empty right child of the node points to its inorder successor.

In general, a threaded binary tree is a binary tree which contains **threads** (i.e.,addresses of some nodes) which facilitate upward movement in the tree.

One way threading:- A thread will appear in a right field of a node and will point to the next node in the inorder traversal.

Two way threading:- A thread will also appear in the left field of a node and will point to the preceding node in the inorder traversal.

One Way Threaded Binary Trees

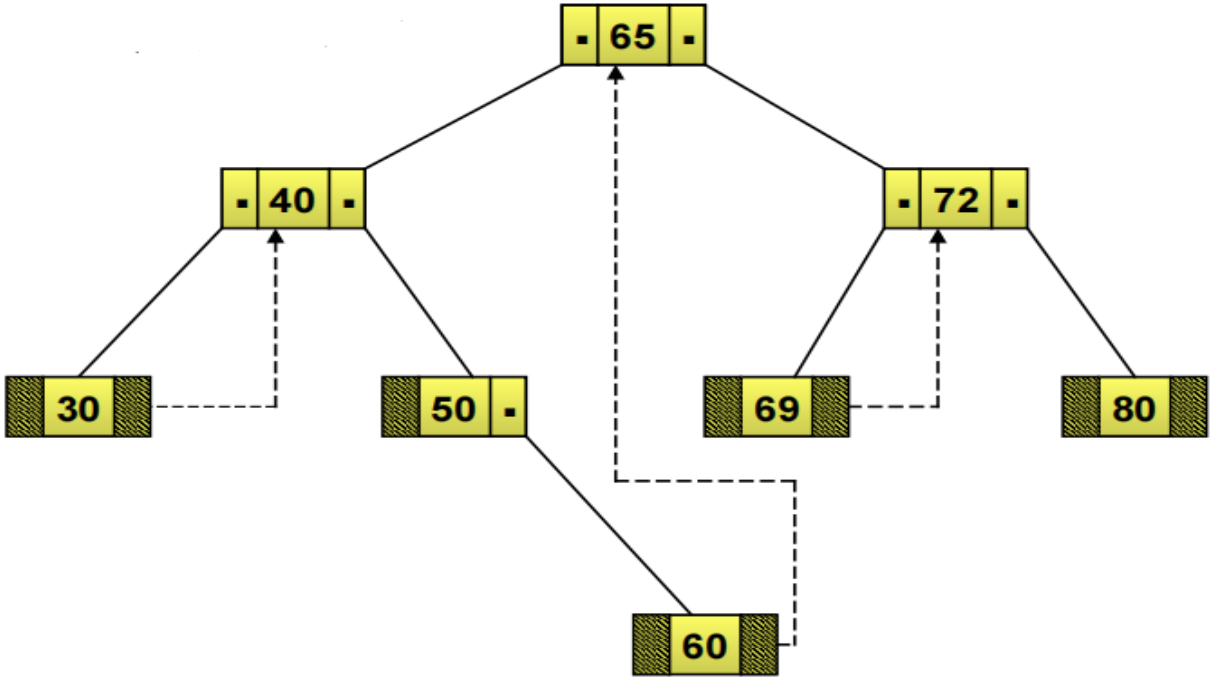
The empty left child field of a node can be used to point to its inorder predecessor.

Similarly, the empty right child field of a node can be used to point to its in-order successor.

Such a type of binary tree is known as a one way threaded binary tree.

A field that holds the address of its in-order successor is known as **thread**

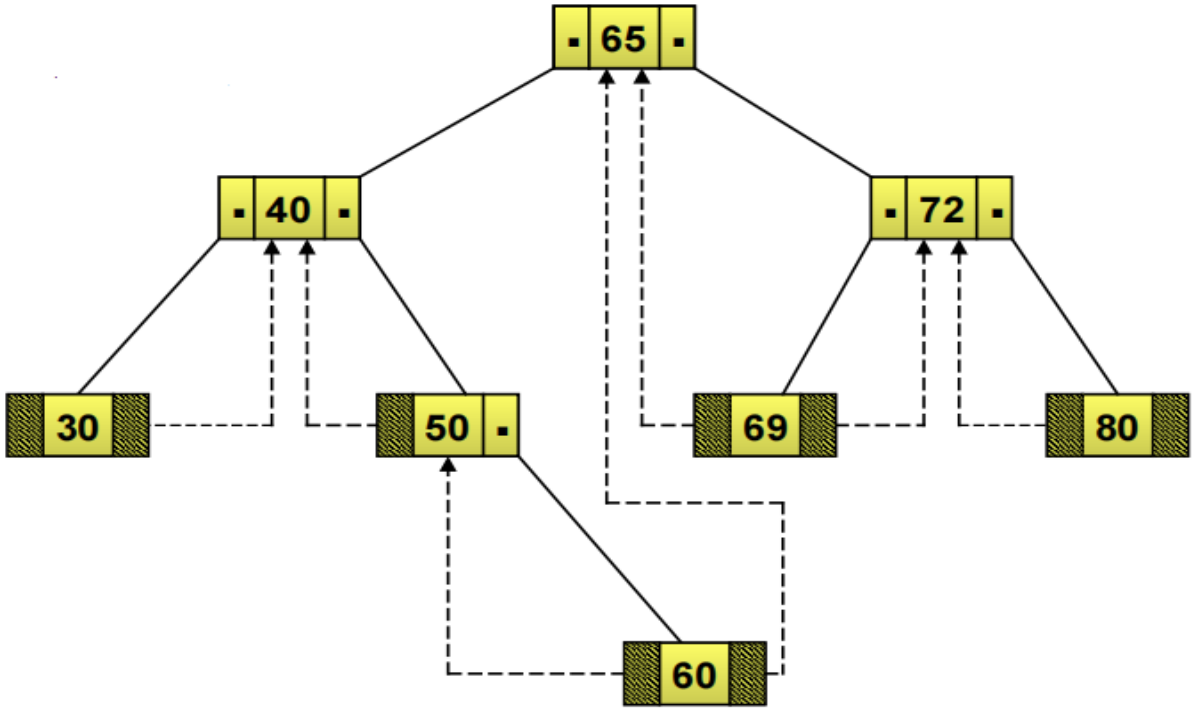
In-order :- 30 40 50 60 65 69 72 80



Two way Threaded Binary Trees

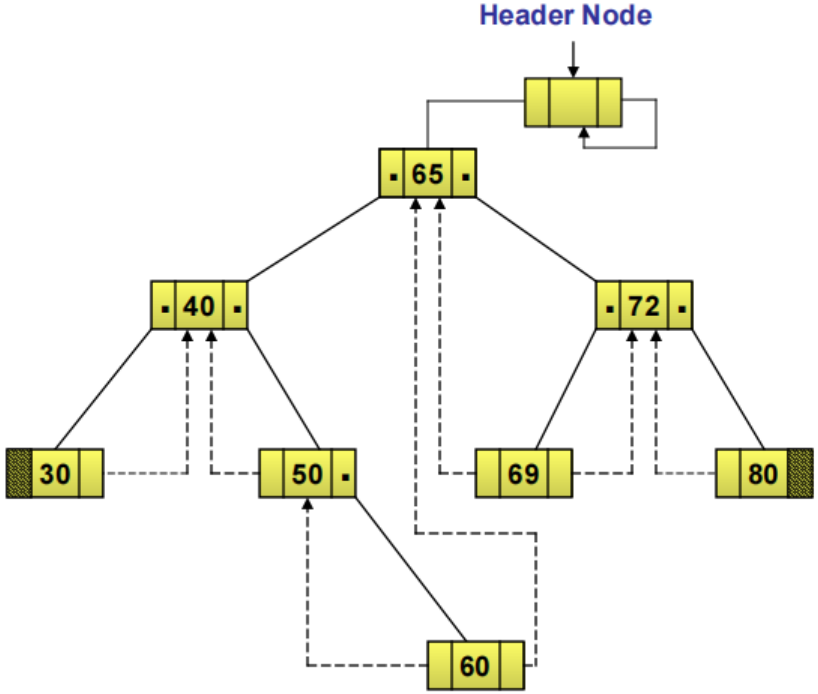
- Such a type of binary tree is known as a threaded binary tree.
- A field that holds the address of its inorder successor or predecessor is known as thread.
- The empty left child field of a node can be used to point to its inorder predecessor.
- Similarly, the empty right child field of a node can be used to point to its inorder successor.

Inorder :- 30 40 50 60 65 69 72 80

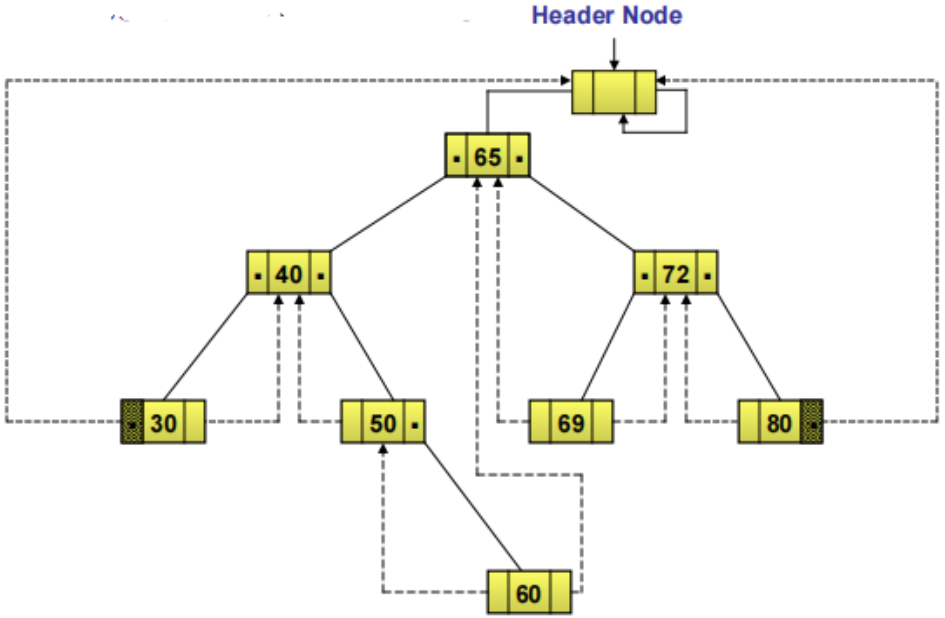


Two way Threaded Binary Trees with header Node

- The right child of the header node always points to itself.
- The threaded binary tree is represented as the left child of the header node.

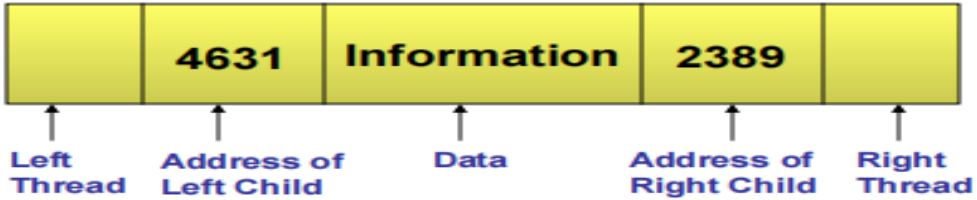


The left thread of node 30 and the right thread of node 80 point to the header node.



Representing a Threaded Binary Tree

- The structure of a node in a threaded binary tree is a bit different from that of a normal binary tree.
- Unlike a normal binary tree, each node of a threaded binary tree contains two extra pieces of information, namely left thread and right thread.
- The left and right thread fields of a node can have two values:
- 1: Indicates a normal link to the child node
- 0: Indicates a thread pointing to the inorder predecessor or inorder successor



A binary tree is threaded based on the traversal technique.

- **In-threaded binary trees**
- Post-threaded binary trees
- Pre-threaded binary trees

When trees are represented in memory, it should be able to distinguish between **threads** and **pointers**.

This can be done byadding two additional fields to node structure, ie., **leftThread** and **rightThread**

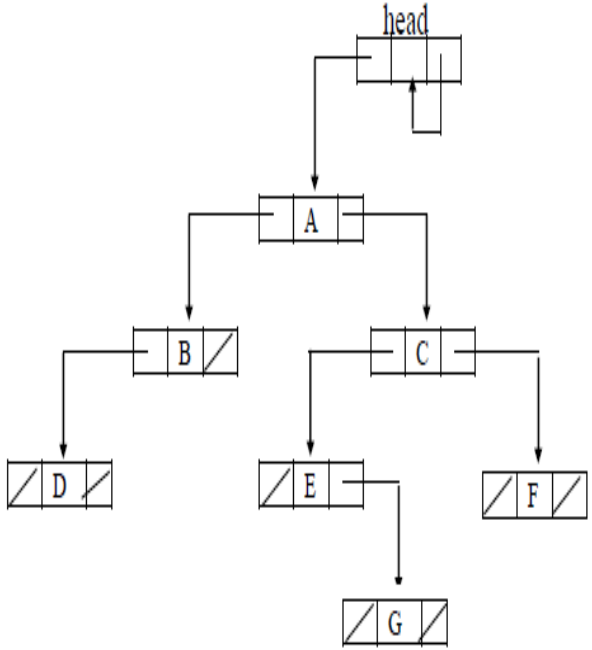
If ptr→leftThread = TRUE, then ptr→leftChild contains a thread, otherwise it contains a pointer to the left child.

If ptr→rightThread = TRUE, then ptr→rightChild contains a thread, otherwise it contains a pointer to the right child.

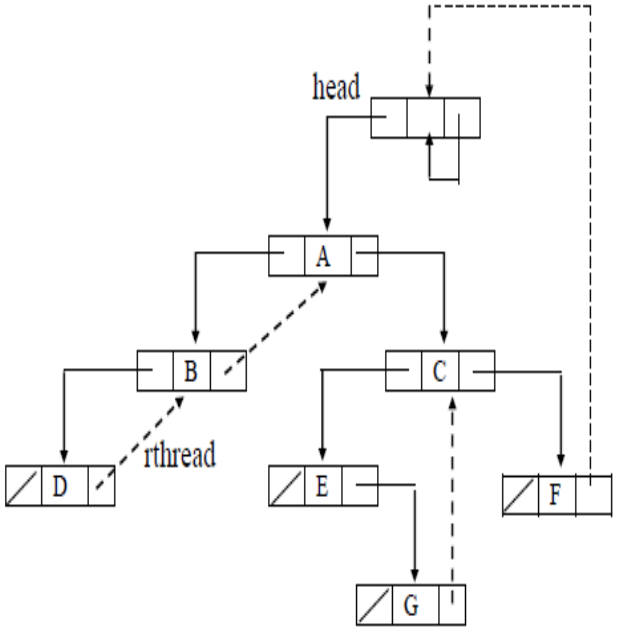
in-threaded binary trees

- In a binary tree, if llink (left link) of any node contains \0 (null) and if it is replaced by address of the inorder predecessor, then the resulting tree is called **left in-threaded binary tree.**
- In a binary tree, if rlink (right link) of a node is NULL and if it is replaced by address of inorder successor, the resulting tree is called **right inthreaded binary tree.**
- An **in-threaded binary tree or inorder threading** of a binary tree is the one which is both left in-threaded and right in-threaded.

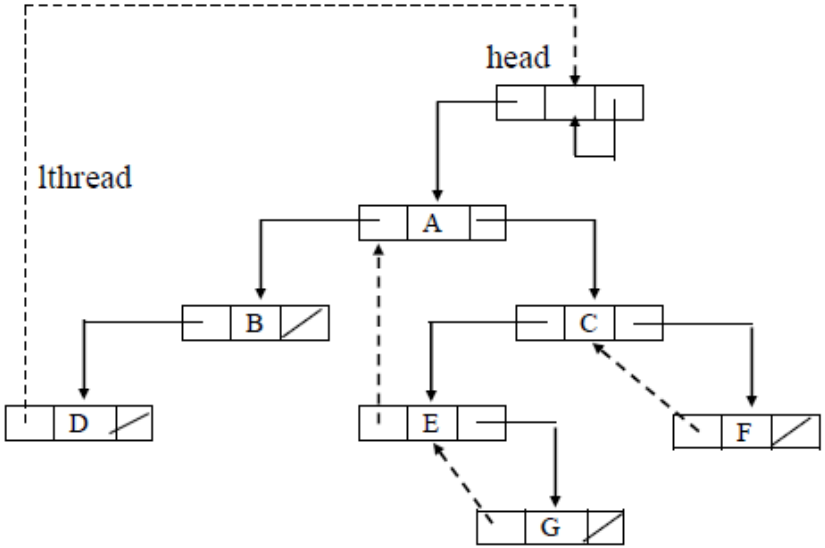
consider the binary tree with header node.

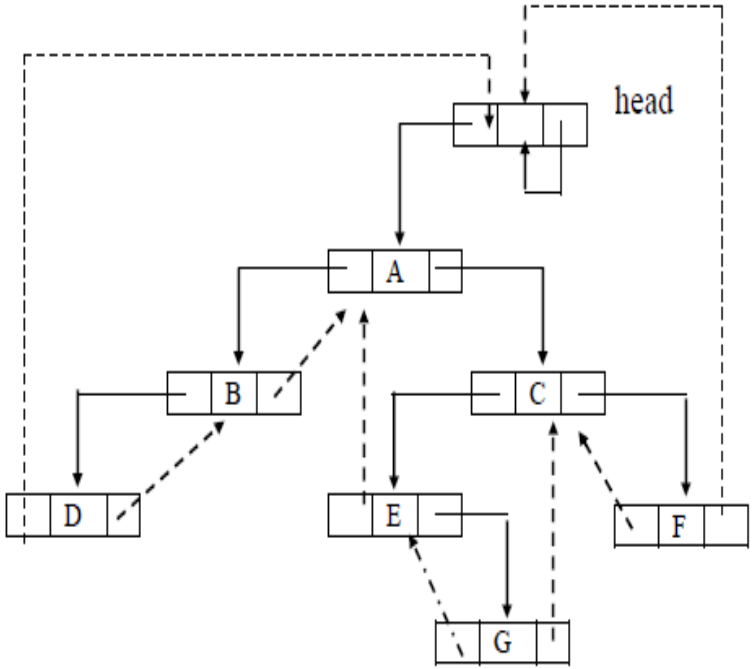


if the right link of a node is NULL and if it is replaced by the address of the inorder successor as shown using dotted lines



- if the left field of a node is NULL and is replaced by the inorder predecessor as shown in fig, then the tree is said to be left in-threaded binary tree.
- Here also an extra field lthread is used where 1 indicates the presence of a thread and 0 indicates ordinary link connecting the left subtree.





In-threaded binary tree (inorder threading of binary tree)

Function to traverse the tree in inorder

```
void inorder(NODE head)
{
NODE temp;
if ( head->llink == head )
{
printf("Tree is empty\n");
return;
}
printf("The inorder traversal of the tree is\n");
temp = head;
for (;;)
{
temp = inorder_successor(temp);
if ( temp == head ) return;
printf("%d ",temp->info);
}
}
```