MODULE 3

LINKED LISTS: Additional List Operations, Sparse Matrices, Doubly Linked List.

TREES: Introduction, Binary Trees, Binary Tree Traversals, Threaded Binary Trees.

Additional List Operations

1. **Inverting a singly linked list:**

```
listpointer invert(listpointer lead)
{/* invert the list pointed to by lead*/
     listpointer middle,trail;
     middle= null;
     while(lead)
     {
       trail=middle
       middle=lead;
       lead=lead->link;
       middle->link=trail;
     }
   return middle;
 }
```

Prg: Inverting a singly linked list

2.**Concatenating singly linked lists:**

```
listpointer concatenate(listpointer ptr1, listpointerptr2)
{  /*produce a new list that contains the list ptr1 followed
   by the list ptr2.the list pointed to by optr1 is changed
   permanently*/

    /*check for empty list*/
    if(!ptr1) return ptr2;
     if(!ptr2) return ptr1;

     /*neither list is empty, find end of first list*/
     for(temp=ptr1;temp->link;temp=temp->link);

     /*link end of first to start of second*/
     temp->link=ptr2;
}
```

Prg: Concatenating singly linked lists

Operations for Circularly Linked Lists:

1.Inserting at front of the list:

```
void insertfront(listpointer *last, listpointer node)
{
     /*insert node at the front of the circular list whose
last node is last*/

  if(!(*last))
  {
      /*list is empty, change last to point to new entry*/
      *last=node;
  }
  else
  {
     /*list is not empty, add new entry at front*/
     node->link=(*last)->link;
     (*last)->link=node;
  }
}
```

Prg: Inserting at the front of a list

To insert at rear , we only need to add the additional statement *last=node to the else clause of insertFront

2. Finding the length of a circular list

```
int length(listpointer last)
{
/*find the length of the circular list last*/

    listpointer temp;
    int count=0;
    if(last)
          {
                              temp=last;
           do
             {
              count++;
              temp=temp->link;
              }
          }
     return count;
 }
```

prg: **finding lrngth of a circular list**

SPARSE MATRIX REPRESENTATON

A linked list representation for sparse matrices.

In data representation, each column of a sparse matrix is represented as a circularly linked list with a header node. A similar representation is used for each row of a sparsematrix. Each node has a tag field, which is used to distinguish between header nodes and entry nodes.

Header Node:

- Each header node has three fields: down, right, and next as shown in figure (a).
- The down field is used to link into a column list and the right field to link into a row list.
- The next field links the header nodes together.
- The header node for row i is also the header node for column i, and the total number of header nodes is max {number of rows, number of columns}.

Element node:

- Each element node has five fields in addition in addition to the tag field: row, col, down, right, value as shown in figure (b).
- The down field is used to link to the next nonzero term in the same column and the right field to link to the next nonzero term in the same row. Thus, if aij ≠ 0, thereis a node with tag field = entry, value = aij, row = i, and col = j as shown in figure (c).
- We link this node into the circular linked lists for row i and column j. Hence, it is simultaneously linked into two different lists.

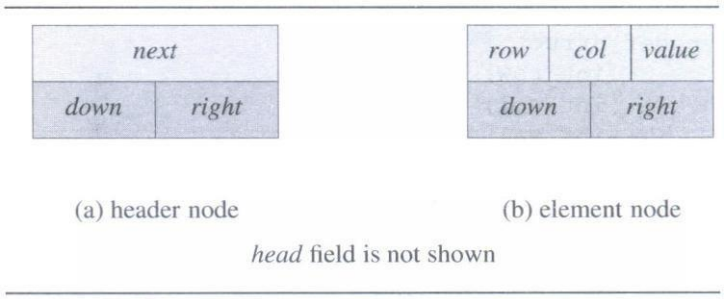


Figure: Node structure for sparse matrices

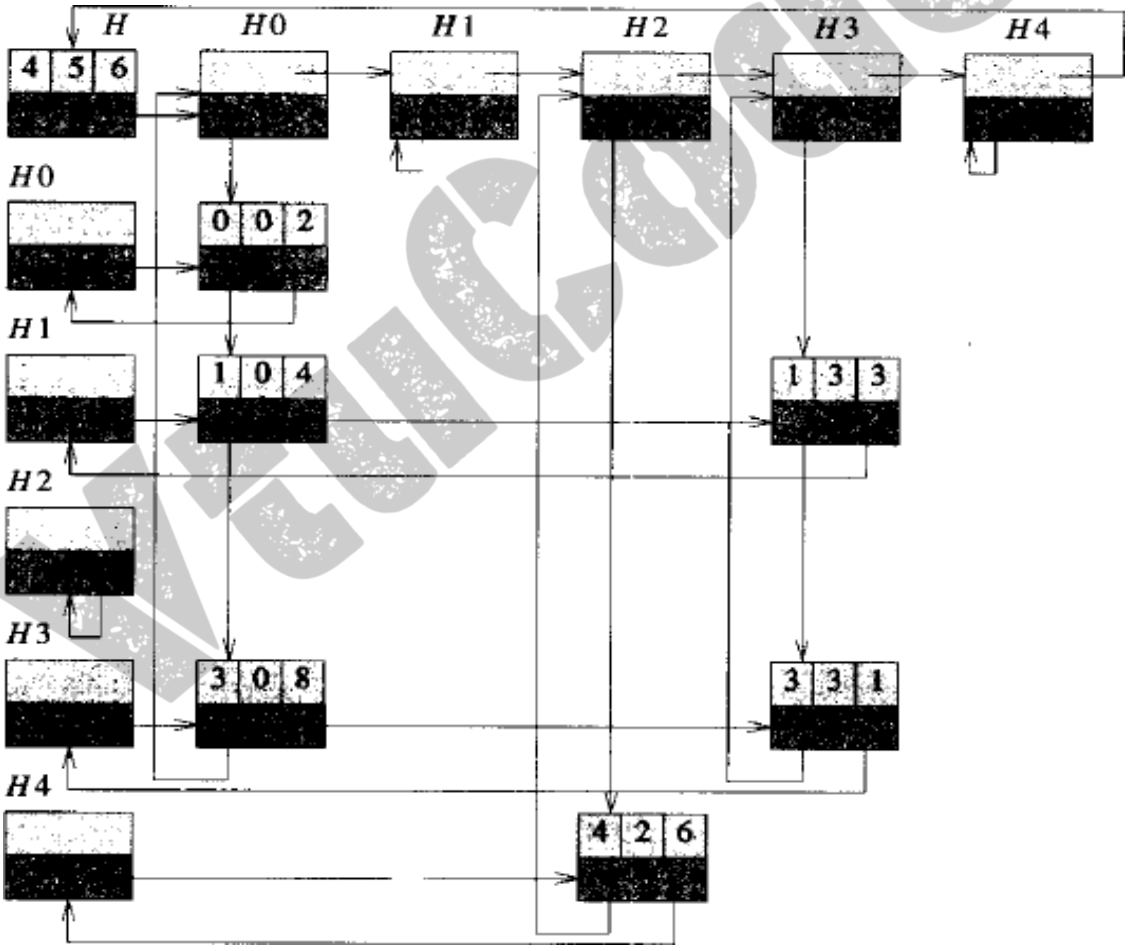
Consider the sparse matrix, as shown in below figure (2).

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

Figure (2): 4×4 sparse matrix a

Figure (3) shows the linked representation of this matrix. Although we have not shownthe value of the tag fields, we can easily determine these values from the node structure.

For each nonzero term of a, have one entry node that is in exactly one row list and one column list. The header nodes are marked HO-H3. As the figure shows, we use the rightfield of the header node list header.



To represent a *numRows x numCols* matrix with *numTerms* nonzero terms, then we need max *{numRows, numCols}* + *numTerms* + *1* node. While each node may require several words of memory, the total storage will be less than *numRows x numCols* when *numTerms* is sufficiently small.

There are two different types of nodes in representation, so unions are used to create the appropriate data structure. The C declarations are as follows:

```
#define MAX_SIZE 50 /*size of largest matrix*/
typedef enum {head,entry} tagfield;
typedef struct matrixNode *matrixPointer;
typedef struct {
        int row;
        int col;
        int value;
        } entryNode;
typedef struct {
        matrixPointer down;
        matrixPointer right;
        tagfield tag;
        union {
           matrixPointer next;
           entryNode entry;
           } u;
        } matrixNode;
matrixPointer hdnode[MAX_SIZE];
```

Doubly Linked List

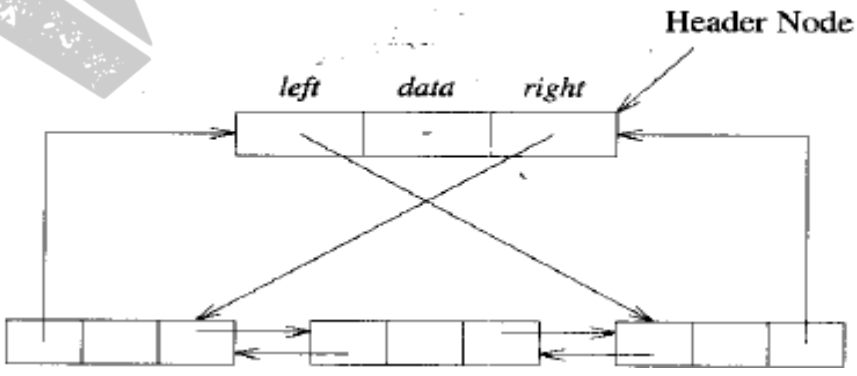
1. The difficulties with single linked lists are that, it is possible to traversal only in one direction, ie., direction of the links.
2. The only way to find the node that precedes p is to start at the beginning of the list. The same problem arises when one wishes to delete an arbitrary node from a singly linked list.Hence the solution is to use doubly linked list

Doubly linked list: It is a linear collection of data elements, called nodes, where each node N isdivided into three parts:

1. An information field INFO which contains the data of N
2. A pointer field LLINK (FORW) which contains the location of the next node in the list
3. A pointer field RLINK (BACK) which contains the location of the preceding node inthe list

A node in a doubly linked list has at least three fields, a left link field (*llink*), a data field (*data*), and a right link field (*rlink*). The necessary declarations are:

```
typedef struct node *nodePointer;
typedef struct {
        nodePointer llink;
        element data;
        nodePointer rlink;
        } node;
```



Insertion into a doubly linked list

Insertion into a doubly linked list is fairly easy. Assume there are two nodes, node and new node,node may be either a header node or an interior node in a list. The function dinsert performs the insertion operation in constant time.

```
void dinsert(nodePointer node, nodePointer newnode)
{/* insert newnode to the right of node */
   newnode→llink = node;
   newnode→rlink = node→rlink;
   node→rlink→llink = newnode;
   node→rlink = newnode;
}
```

Insertion into a doubly linked circular list

Deletion from a doubly linked list

Deletion from a doubly linked list is equally easy. The function *ddelete* deletes the node deletedfrom the list pointed to by node.

To accomplish this deletion, we only need to change the link fields of the nodes that precede(deleted→llink→rlink) and follow (deleted→rlink→llink) the node we want to delete.

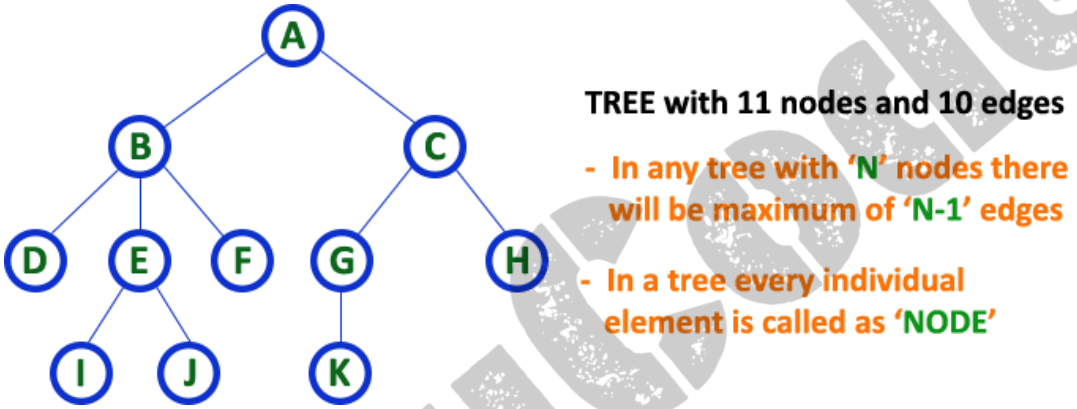
```
void ddelete(nodePointer node, nodePointer deleted)
{/* delete from the doubly linked list */
   if (node == deleted)
      printf("Deletion of header node not permitted.\n");
   else {
      deleted→llink→rlink = deleted→rlink;
      deleted→rlink→llink = deleted→llink;
      free(deleted);
   }
}
```

Deletion from a doubly linked circular list

TREES

- In linear data structure, data is organized in sequential order and in non-linear data structure, data is organized in random order. Tree is a very popular data structure used in wide range of applications. A tree data structure can be defined as follows...
- Tree is a non-linear data structure which organizes data in hierarchical fashion and the tree structure follows a recursive pattern of organizing and storing data.
- Every individual element is called as Node. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure.
- if there are N number of nodes in a tree structure, then there can be a maximum of N-1 number of links.

Example



DEFINITION

A *tree* is a finite set of one or more nodes such that

- There is a specially designated node called *root*.
- The remaining nodes are partitioned into $n >= 0$ disjoint set $T1,\ldots,Tn$, where each of these sets is a tree. $T1,\ldots,Tn$ are called the *subtrees* of the root.

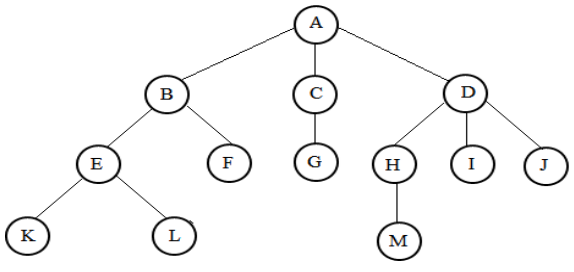


Figure (A)

Every node in the tree is the root of some subtree.

1. **Root**

In a tree data structure, the first node is called as Root Node. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree. **Ex: 'A' in the above tree**

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. Ex: Line between two nodes.

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".

Ex: A,B,C,E & G are parent nodes

4. Child

In a tree data structure, the node which is descendant of any node is called as CHILD Node. In simple words, 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.Ex: B & C are children of A, G & H are children of C and K child of G

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as SIBLINGS. In simple words, the nodes with same parent are called as Sibling nodes.Ex: B & C are siblings, D, E and F are siblings, G & H are siblings, I & J are siblings

6. Leaf

In a tree data structure, the node which does not have a child is called as LEAF Node. In simple words, a leaf is a node with no child. In a tree data structure, the leaf nodes are also called as External Nodes. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node.Ex: D,I,J,F,K AND Hare leaf nodes

7. Internal Nodes

In a tree data structure, the node which has atleast one child is called as INTERNAL Node. In simple words, an internal node is a node with atleast one child.

In a tree data structure, 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.

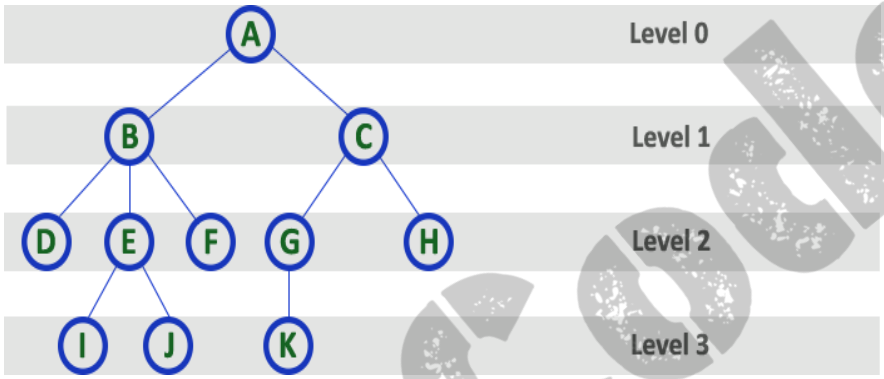
Ex: A,B,C,E & G

8. Degree of a node

In a tree data structure, the total number of children of a node is called as DEGREE of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as **'Degree of Tree'.**Ex: Degree of B is 3, A is 2 and of F is 0

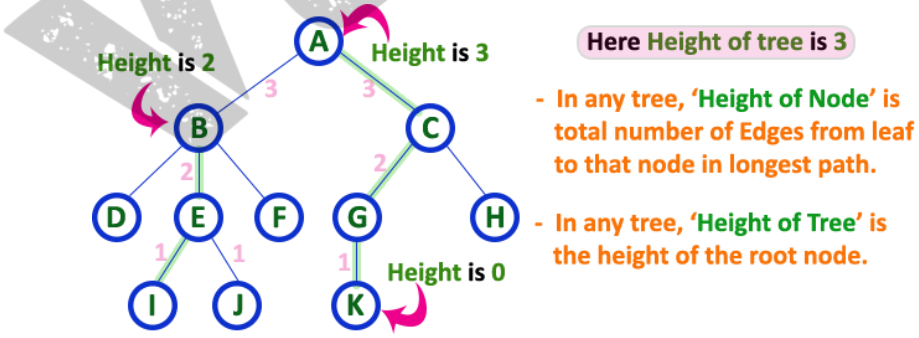
9. Level of a node

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 simple words, 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).



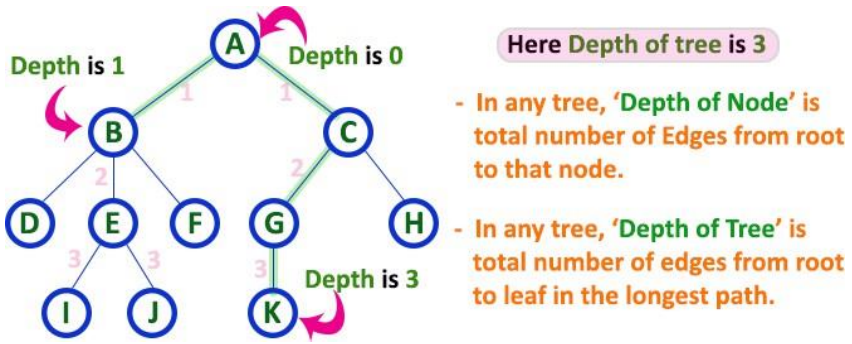
10. Height

In a tree data structure, 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. In a tree, height of all leaf nodes is '0'.



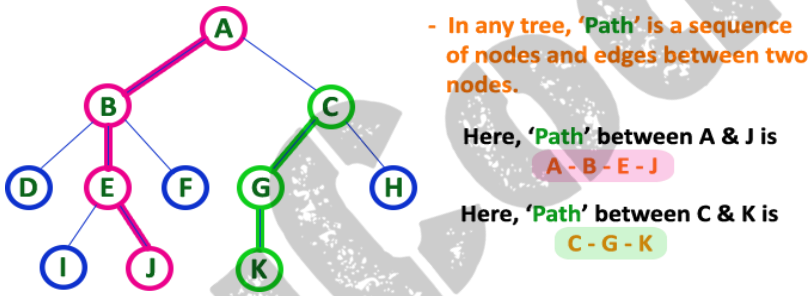
11. Depth

In a tree data structure, the total number of egdes from root node to a particular node is called as DEPTH of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be Depth of the tree. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, depth of the root node is '0'.



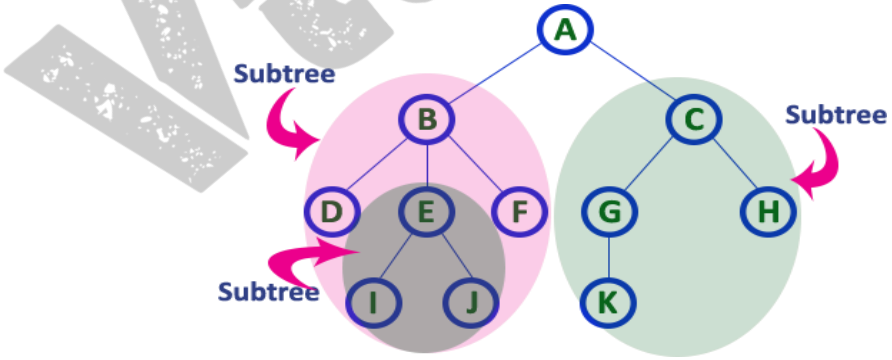
12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between the 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.



13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



The ***ancestors*** of a node are all the nodes along the path from the root to that node.

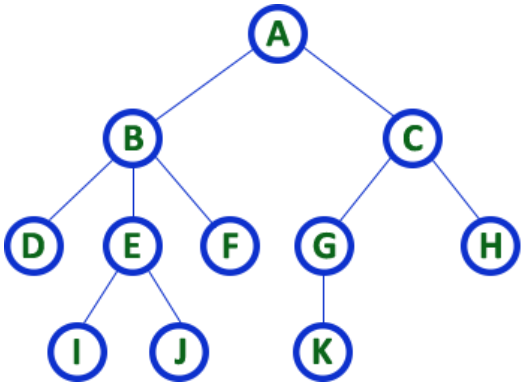
Ex: ancestor of j is B & A

A ***forest*** is a set of *n* 0 disjoint trees. The notion of a forest is very close to that of a tree because if we remove the root of a tree we get a forest. For example, in figure 1 if we remove *A* we get a forest with three trees.

General Tree Representations

A general Tree Structure can be represented with the following three methods. Those methods are as follows...

1. List Representation
2. Left Child - Right Sibling Representation
3. Degree two Representation (Binary Tree Representation) Consider the following tree...



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'

1. List Representation

There are several ways to draw a tree. One useful way is as a list. The tree in the above figure could be written as the list **(A(B(D,E(I,J),F),C(G(K),H)) – list representation (with rounded brackets).**The information in the root node comes first followed by a list of the subtrees of that node.Now, how do we represent a tree in memory? If we wish to use linked lists, then a node must have a varying number of fields depending upon the number of branches.

Possible node structure for a tree of degree k called k-ary tree

Data	link1	link2	.....	link k

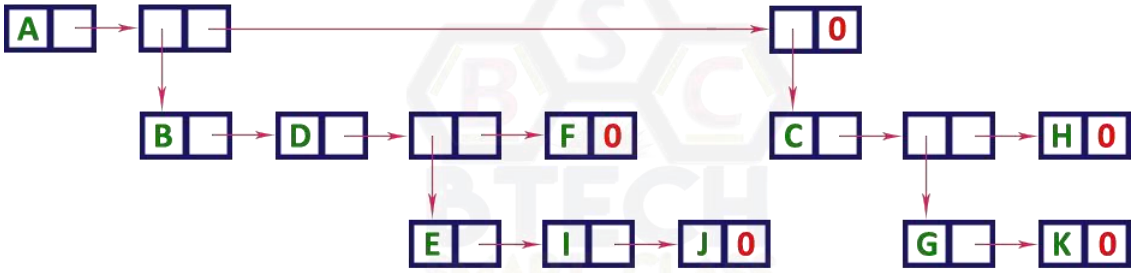
Each link field is used to point to a subtree. This node structure is cumbersome for the following reasons (1) Multiple node structure for different tree nodes (2) Waste of space (3) Excessive use of links.

The other alternate method is to have linked list of child nodes which allocates memory only for the nodes which have children.

In this representation, we use two types of nodes, one for representing the node with data and another for representing only references. We start with a node with data from root node in the tree. Then it is linked to an internal node through a reference node and is linked to any other

node directly. This process repeats for all the nodes in the tree.

The above tree example can be represented using List representation as follows...



2. Left Child - Right Sibling Representation

In this representation, we use list with one type of node which consists of three fields namely Data field, Left child reference field and Right sibling reference field.

To convert the tree of Figure (A) into this representation:

1. First note that every node has at most one leftmost child
2. At most one closest right sibling.

- In Figure (A), the leftmost child of A is B, and the leftmost child of D is H.
- The closest right sibling of B is C, and the closest right sibling of H is I.
- Choose the nodes based on how the tree is drawn. The left child field of each node points to its leftmost child (if any), and the right sibling field points to its closest right sibling (if any).
- Figure (D) shows the tree of Figure (A) redrawn using the left child-right sibling representation.

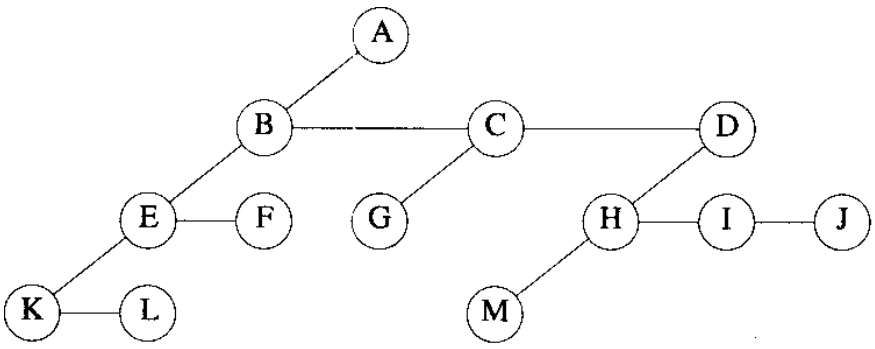


Figure (D): Left child-right sibling representation of tree of figure (A)

3. Degree two tree (Binary Tree)

To obtain the degree-two tree representation of a tree, simply rotate the right-sibling pointers in a left child-right sibling tree clockwise by 45 degrees. This gives us the degree-two tree displayed in Figure (E).

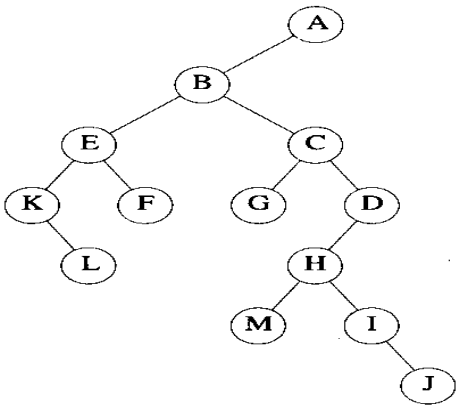


Figure (E): degree-two representation

In the degree-two representation, a node has two children as the left and right children.

BINARY TREES

Definition: A binary tree T is defined as a finite set of nodes such that,

- T is empty or
- T consists of a root and two disjoint binary trees called the left subtree and the right subtree.

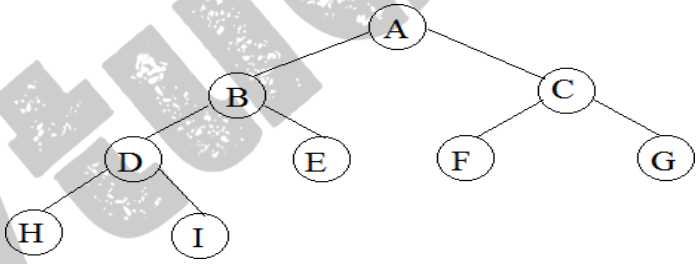


Figure: Binary Tree

Different kinds of Binary Tree

1. Skewed Tree

A skewed tree is a tree, skewed to the left or skews to the right.

or

It is a tree consisting of only left subtree or only right subtree.

- A tree with only left subtrees is called Left Skewed Binary Tree.
- A tree with only right subtrees is called Right Skewed Binary Tree.

2. Complete Binary Tree

A binary tree T is said to complete if all its levels, except possibly the last level, have the maximum number node 2^i, $i \geq 0$ and if all the nodes at the last level appears as far left as possible.

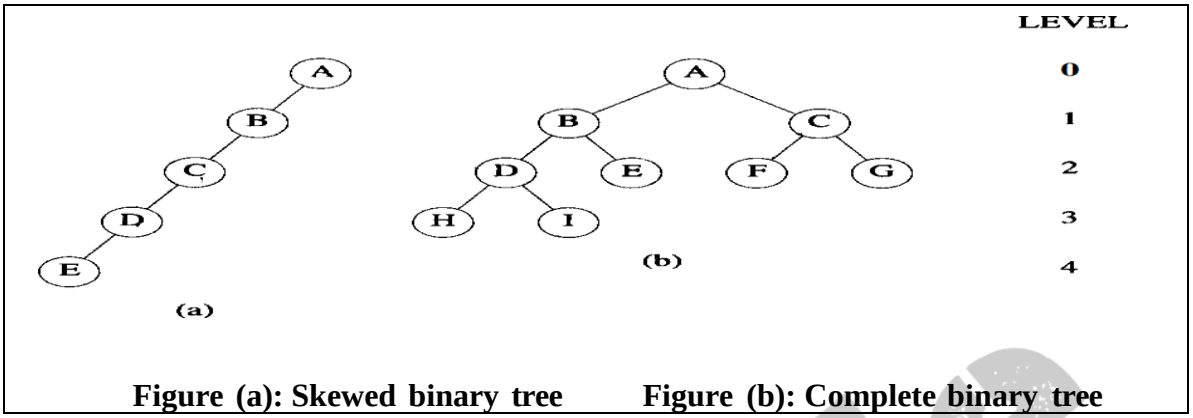


Figure (a): Skewed binary tree **Figure (b): Complete binary tree**

3.Full Binary Tree

A full binary tree of depth 'k' is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 1$.

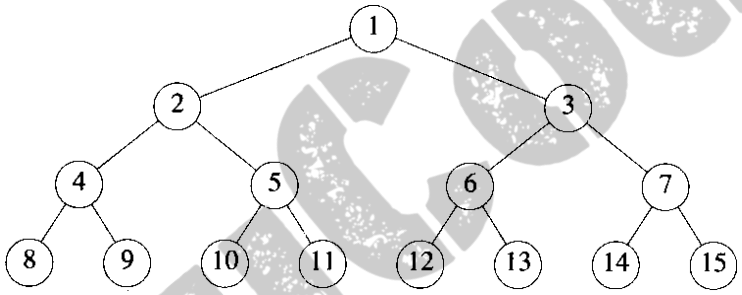
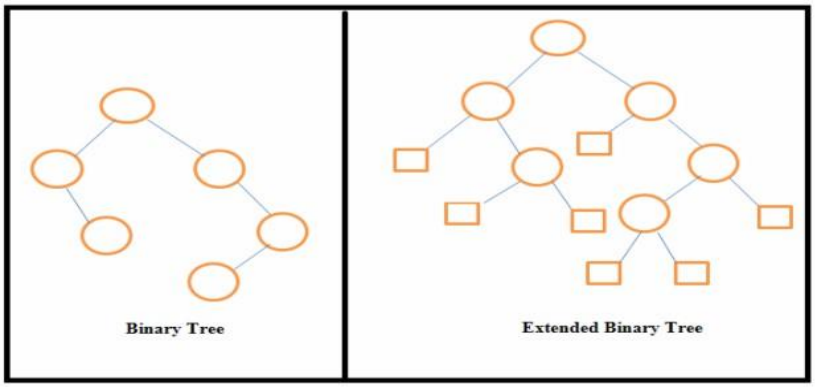


Figure: Full binary tree of level 4 with sequential node number

4. Extended Binary Trees or 2-trees

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

For instance, consider the following binary tree.



The following tree is its extended binary tree. The circles represent internal nodes, and square represent external nodes.

Every internal node in the extended tree has exactly two children, and every external node is a leaf. The result is a complete binary tree.

PROPERTIES OF BINARY TREES

Lemma 1: [Maximum number of nodes]:

(1) The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.

(2) The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.

Proof:

(1) **The proof is by induction on i.**

Induction Base: The root is the only node on level i = 1. Hence, the maximum number of nodes on level i =1 is $2^{i-1} = 2^0 = 1$.

Induction Hypothesis: Let i be an arbitrary positive integer greater than 1. Assume that the maximum number of nodes on level i -1is 2^{i-2}

Induction Step: The maximum number of nodes on level i -1 is 2^{i-2} by the induction hypothesis. Since each node in a binary tree has a maximum degree of 2, the maximum number of nodes on level i is two times the maximum number of nodes on level i-1, or 2^{i-1}

(2) The maximum number of nodes in a binary tree of depth k is

$$\sum_{i=0}^{k} (\text{maximum number of nodes on level } i) = \sum_{i=0} 2^{i-1} = 2^{k-1}$$

Lemma 2: [Relation between number of leaf nodes and degree-2 nodes]:

For any nonempty binary tree, T, if n0 is the number of leaf nodes and n2 the number of nodes of degree 2, then n0 = n2 + 1.

Proof: Let n_1 be the number of nodes of degree one and n the total number of nodes. Since all nodes in T are at most of degree two, we have

$$n = n0 + n1 + n2 \quad (1)$$

Count the number of branches in a binary tree. If B is the number of branches,

then n =B + 1.

All branches stem from a node of degree one or two. Thus,

$$B = n_1 + 2n_2.$$

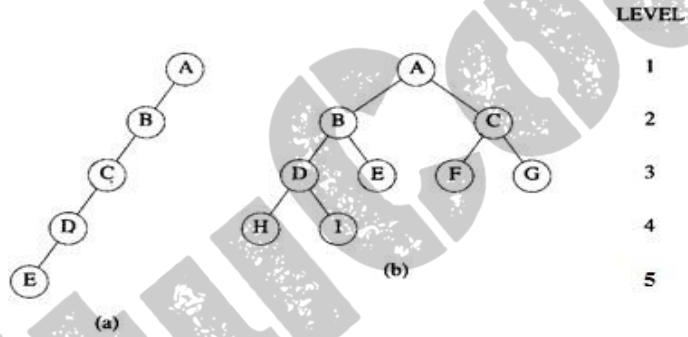
Hence, we obtain

$$n = B + 1 = n_1 + 2n_2 + 1 \quad (2)$$

Subtracting Eq. (2) from Eq. (1) and rearranging terms, we get

$$\mathbf{n_0 = n_2 + 1}$$

Consider the figure:



Here, For Figure (b) n_2=4, $n_0 = n_2 + 1 = 4 + 1 = 5$ Therefore, the total number of leaf node=5

BINARY TREE REPRESENTATION

The storage representation of binary trees can be classified as

1. Array representation
2. Linked representation.

Array representation:

- A tree can be represented using an array, which is called sequential representation.
- The nodes are numbered from 1 to n, and one dimensional array can be used to store the nodes.
- Position 0 of this array is left empty and the node numbered *i* is mapped to position *i* of the array.

  Below figure shows the array representation for both the trees of figure (a).

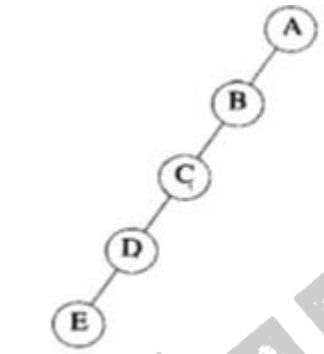


Figure 1(a) Skewed binary tree

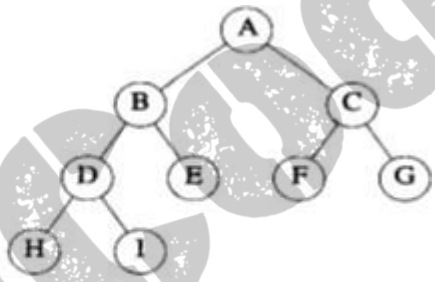


Figure 1(b) Complete binary tree

	tree
[0]	–
[1]	A
[2]	B
[3]	–
[4]	C
[5]	–
[6]	–
[7]	–
[8]	D
[9]	–
.	.
.	.
.	.
[16]	E

(a). Tree of figure 1(a)

	tree
[0]	–
[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I
.	.
.	.
.	.
[16]	

(b). Tree of figure 1(b)

- For complete binary tree the array representation is ideal, as no space is wasted.
- For the skewed tree less than half the array is utilized.

Linked representation:

The problems in array representation are:

- It is good for complete binary trees, but more memory is wasted for skewed and many other binary trees.
- The insertion and deletion of nodes from the middle of a tree require the movement of many nodes to reflect the change in level number of these nodes.

These problems can be easily overcome by linked representation

Each node has three fields,

- LeftChild - which contains the address of left subtree
- RightChild - which contains the address of right subtree.
- Data - which contains the actual information

C Code for node:

```
typedef struct node *treepointer;
typedef struct
{
      int data;
      treepointer leftChild, rightChild
}node;
```

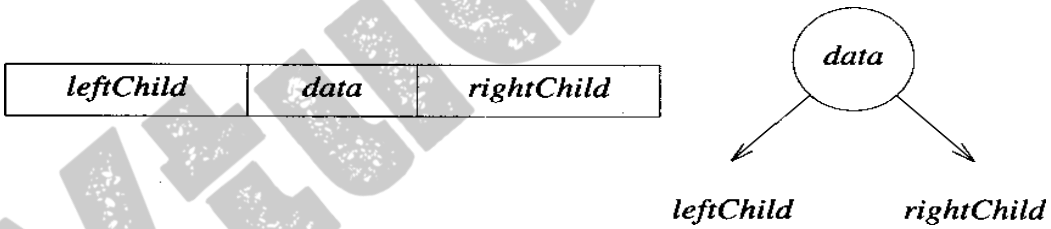


Figure: Node representation

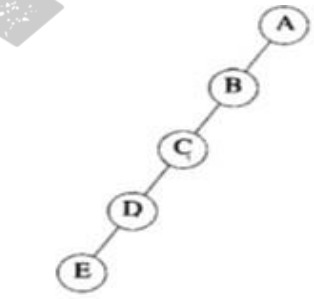


Figure 1(a) Skewed binary tree

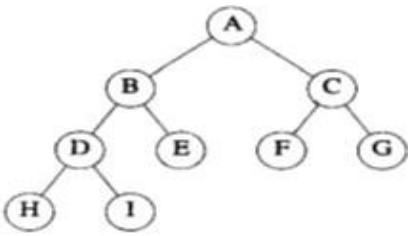


Figure 1(b) Complete binary tree

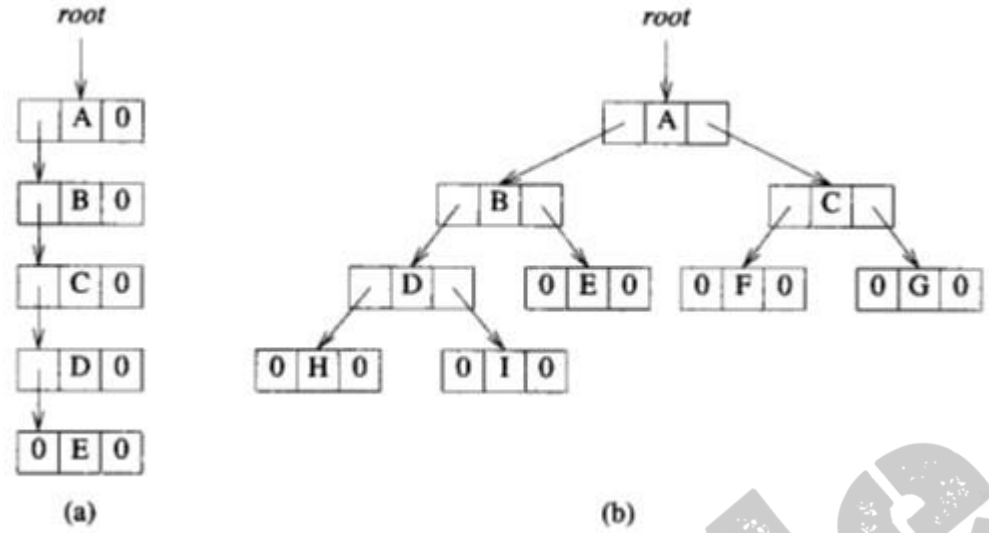


Figure: Linked representation of the binary tree

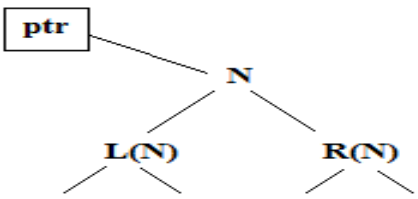
BINARY TREE TRAVERSALS

Visiting each node in a tree exactly once is called tree traversal The different methods of traversing a binary tree are:

1. Preorder
2. Inorder
3. Postorder
4. Iterative inorder Traversal
5. Level-Order traversal

1. Inorder: Inorder traversal calls for moving down the tree toward the left until you cannot go further. Then visit the node, move one node to the right and continue. If no move can be done, then go back one more node.

Let ptr is the pointer which contains the location of the node N currently being scanned. L(N) denotes the leftchild of node N and R(N) is the right child of node N



Recursion function:

The inorder traversal of a binary tree can be recursively defined as

- Traverse the left subtree in inorder.
- Visit the root.
- Traverse the right subtree in inorder.

```
void inorder(treepointerptr)
{
      if (ptr)
      {
           inorder (ptr→leftchild);
           printf ("%d",ptr→data);
           inorder (ptr→rightchild);
        }
}
```

2.Preorder: Preorder is the procedure of visiting a node, traverse left and continue. When you cannot continue, move right and begin again or move back until you can move right and resume.

Recursion function:

The Preorder traversal of a binary tree can be recursively defined as

- Visit the root
- Traverse the left subtree in preorder.
- Traverse the right subtree in preorder

```
void preorder (treepointerptr)
{
     if (ptr)
     {
          printf ("%d",ptr→data)
          preorder (ptr→leftchild);
          preorder (ptr→rightchild);
    }
 }
```

3. Postorder: Postorder traversal calls for moving down the tree towards the left until you can go no further. Then move to the right node and then visit the node and continue.

Recursion function:

The Postorder traversal of a binary tree can be recursively defined as

- Traverse the left subtree in postorder.
- Traverse the right subtree in postorder.
- Visit the root

```
void postorder(treepointerptr)
{
      if (ptr)
      {
       postorder
       (ptr→leftchild);
       postorder
       (ptr→rightchild);
       printf
       ("%d",ptr→data);
       }
  }
```

4. 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, and the node's right child is stacked until a null node is reached. The traversal then continues with the left child. The traversal is complete when the stack is empty.

```
void iterInorder(treePointer node)
{
   int top = -1; /* initialize stack */
   treePointer stack[MAX_STACK_SIZE];
   for (;;) {
      for(; node; node = node→leftChild)
         push(node); /* add to stack */
      node = pop(); /* delete from stack */
      if (!node) break; /* empty stack */
      printf("%d", node→data);
      node = node→rightChild;
   }
}
```

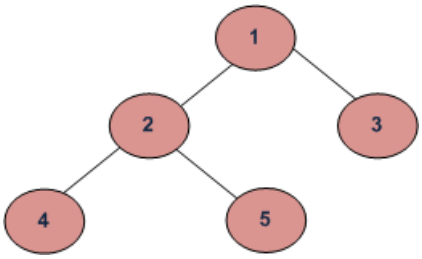
Program : Iterative inorder traversal

5. Level-Order traversal:

Visiting the nodes using the ordering suggested by the node numbering is called level ordering traversing.

The nodes in a tree are numbered starting with the root on level 1 and so on.

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.

Function for level order traversal of a binary tree:

```
void levelOrder(treePointer ptr)
{/* level order tree traversal */
   int front = rear = 0;
   treePointer queue[MAX_QUEUE_SIZE];
   if (!ptr) return; /* empty tree */
   addq(ptr);
   for (;;) {
      ptr = deleteq();
      if (ptr) {
         printf("%d",ptr→data);
         if(ptr→leftChild)
            addq(ptr→leftChild);
         if (ptr→rightChild)
            addq(ptr→rightChild);
      }
      else break;
   }
}
```

Program : Level-order traversal of a binary tree

THREADED BINARY TREE

The limitations of binary tree are:

- In binary tree, there are n+1 null links out of 2n total links.
- Traversing a tree with binary tree is time consuming. These limitations can be overcome by threaded binary tree.

In the linked representation of any binary tree, there are more null links than actual pointers. These null links are replaced by the pointers, called **threads**, which points to other nodes in the tree.

To construct the threads use the following rules:

1. Assume that **ptr** represents a node. If ptr→leftChild is null, then replace the null link with a pointer to the inorder predecessor of ptr.
2. If ptr →rightChild is null, replace the null link with a pointer to the inorder successor of ptr.

Ex: Consider the binary tree as shown in below figure:

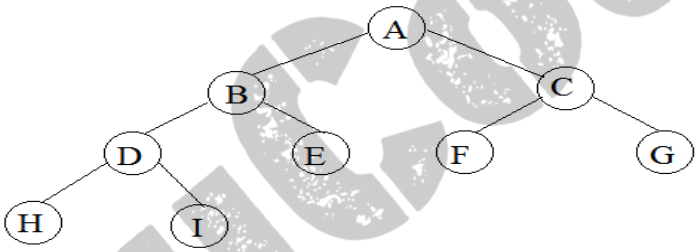


Figure A: Binary Tree

There should be no loose threads in threaded binary tree. But in **Figure B** two threads have been left dangling: one in the left child of *H,* the other in the right child of G.

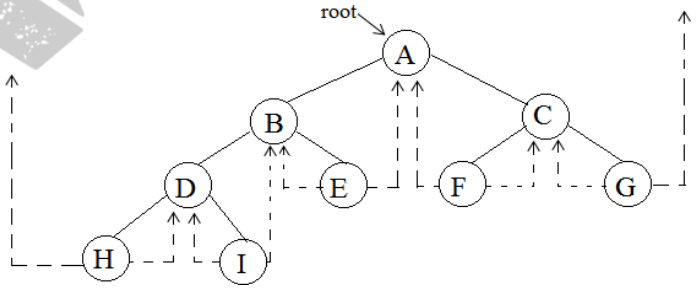


Figure B: Threaded tree corresponding to Figure A

In above figure the new threads are drawn in broken lines. This tree has 9 node and 10 0-links which has been replaced by threads.

When trees are represented in memory, it should be able to distinguish between threads and pointers. This can be done by adding 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.

Node Structure:

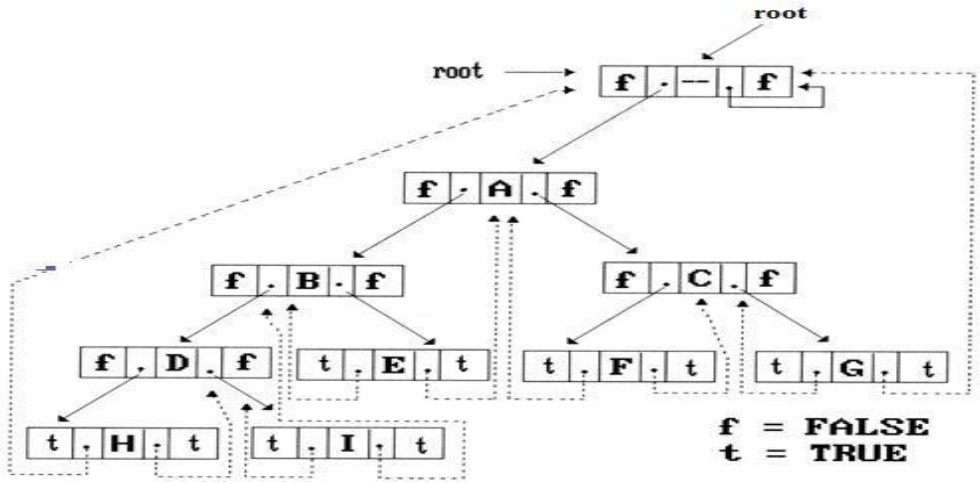
The node structure is given in C declaration

```
typedef struct threadTree *threadPointer
typedef struct{
                short int leftThread; threadPointer
                leftChild;
                char data;
                threadPointer rightChild;
                short int rightThread;
          }threadTree;
```

Figure An empty threaded tree

The complete memory representation for the tree of figure is shown in Figure C

The variable ***root*** points to the header node of the tree, while **root →leftChild** points to the start of the first node of the actual tree. This is true for all threaded trees. Here the problem of the loose threads is handled by pointing to the head node called ***root***.



Inorder Traversal of a Threaded Binary Tree

- By using the threads, 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 any node in a threaded tree without using a stack.

```
threadedpointer insucc(threadedPointer tree)
{    /* find the inorder successor of tree
     in a threaded binary tree */
     threadedpointer temp;
     temp = tree→rightChild;
     if (!tree→rightThread)
       while (!temp→leftThread)
              temp = temp→leftChild;
     return temp; }
```

Program: Finding inorder successor of a node

To perform inorder traversal make repeated calls to insucc ( ) function

```
void tinorder (threadedpointer tree)
{
Threadedpointer temp = tree;
for(; ;){
            temp = insucc(temp);
            if (temp == tree) break;
            printf("%3c", temp→data
    }
}
```

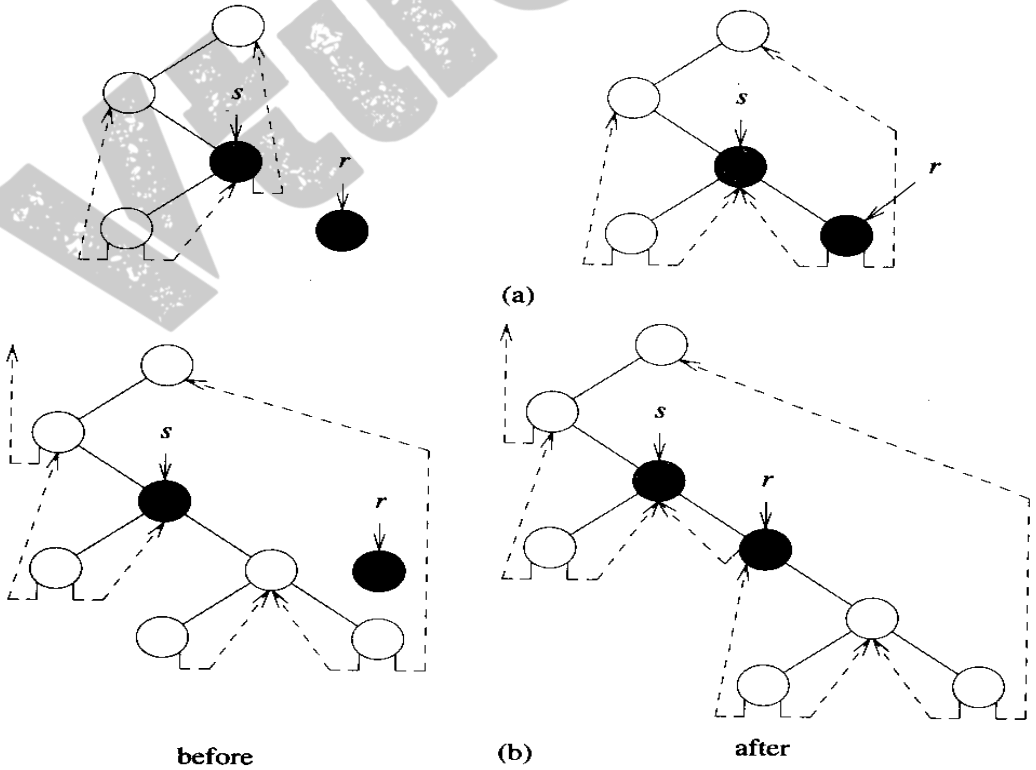
Program: Inorder traversal of a threaded binary tree

Inserting a Node into a Threaded Binary Tree

In this case, the insertion of **r** as the right child of a node **s** is studied.

The cases for insertion are:

- If ***s*** has an **empty** right subtree, then the insertion is simple and diagrammed in Figure
- If the right subtree of ***s*** is **not empty**, then this right subtree is made the right subtree of ***r*** after insertion. When this is done, r becomes the inorder predecessor of a node that has a **leftThread == true** field, and consequently there is a thread which has to be updated to point to r. The node containing this thread was previously the inorder successor of s.



```
void insertRight(threadedPointer Sf, threadedPointer r)
{  /* insert r as the right child of s */
     threadedpointer temp;
     r→rightChild = parent→rightChild;
     r→rightThread = parent→rightThread;
     r→leftChild = parent;
     r→leftThread = TRUE;
     s→rightChild = child;
     s→rightThread = FALSE;
     if (!r→rightThread)
     {
          temp = insucc(r);
          temp→leftChild = r;
      }
}
```