Rashtreeya Sikshana Samithi Trust

RV Institute of Technology and Management®

(Affiliated to VTU, Belagavi)

JP Nagar, Bengaluru - 560076

Department Computer Science and Engineering

&

Department Information Science and Engineering

Course Name : DATA STRUCTURES AND APPLICATIONS

Course Code : BCS304

III Semester

2022 Scheme

Module-3

Linked Lists and Trees

3.1 ADDITIONAL LIST OPERATIONS

3.1.1 Operations for Chains

It is often necessary, and desirable, to build a variety of functions for manipulating singly linked lists. Inverting (or reversing) a chain is another useful operation. This routine is especially interesting because we can do it "in place" if we use three pointers. We use the following declarations:

```
typedef struct list—node *list_pointer;
typedef struct list—node
{
char data;
list_pointer link;
};



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

Another useful function is one that concatenates two chains, ptr1 and ptr2.

```
list—pointer concatenate(list—pointer ptrl, list—pointer ptr2)
{
/* produce a new list that contains the list ptrl followed
by the list ptr2. The list pointed to by ptrl is changed permanently */

list—pointer temp;
if (IS—EMPTY(ptrl)) return ptr2;
else {
if (!IS—EMPTY(ptr2))
{
for (temp = ptrl; temp->link; temp = temp->link)
;
temp->link = ptr2;
)
return ptrl;
}
}
```

3.1.2 Operations for Circularly Linked Lists

Now let us take another look at circular lists like the one in Figure 4.16. Suppose we want to insert a new node at the front of this list. We have to change the link field of the node containing x3. This means that we must move down the entire length of a until we find the last node. It is more convenient if the name of the circular list points to the last node rather than the first. Now we can write functions that insert a node at the front or at the rear of a circular list in a fixed amount of time. To insert node at the rear, we only need to add the additional statement *ptr= node to the else clause of insert-front.

```
void insert—front(list—pointer *ptr, list—pointer node)
/* insert node at the front of the circular list ptr, where ptr is the last node in
the list */
{

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

Finding the length of a circular list.

```
int length(list—pointer ptr)
{
/* find the length of the circular list ptr */
list—pointer temp;
int count = 0;
if (ptr) {
temp = ptr;
do {
count++;
temp = temp->link;
} while {temp != ptr);
}
return count;
)
```

3.1.3 SPARSE MATRICES

In this section, we study a linked list representation for sparse matrices. As we have seen previously, linked lists allow us to efficiently represent structures that vary in size, a benefit that also applies to sparse matrices.

In our data representation, we represent each column of a sparse matrix as a circularly linked list with a head node. We use a similar representation for each row of a sparse matrix. Each node has a tag field, which we use to distinguish between head nodes and entry nodes. Each head node has three additional fields: down, right, and next. We use the down field to link into a column list and the right field to link into a row list. The next field links the head nodes together. The head node for row i is also the head node for column i, and the total number of head nodes is max {number of rows, number of columns}.

Each entry node has five fields in addition to the tag field: row, col, down, right, value (Figure 4.19(b)). We use the down field 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 $a_{ij} \neq 0$, there is a node with tag field = entry, value = a_{ij}, row = i, and col = j as shown in Fig 3.1. We link this node into the circular linked lists for row i and column j. Hence, it is simultaneously linked into two different lists.

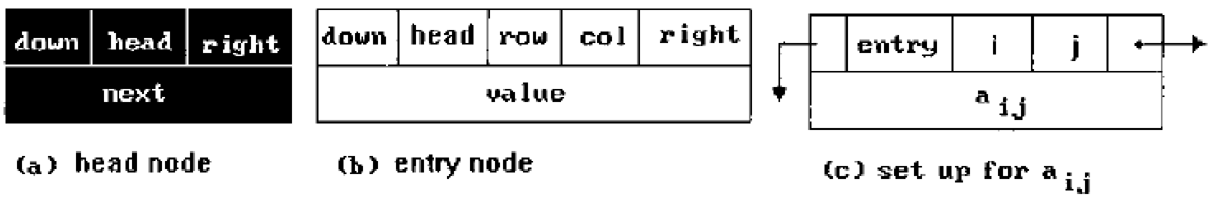


Fig 3.1: Sparse matrix representation

The list of head nodes also has a head node that has the same structure as an entry node. We use the row and col fields of this node to store the matrix dimensions.

Suppose that we have the sample sparse matrix, a, shown in Fig 3.2. Fig 3.3 shows the linked representation of this matrix. Although we have not shown the value of the tag fields, we can easily determine these values from the node structure. For each nonzero term of a, we have one entry node that is in exactly one row list and one column list. The head nodes are marked H0-H3. As the figure shows, we use the right field of the head node list header to link into the list of head nodes. Notice also that we may reference the entire matrix through the head node, a, of the list of head nodes.

$$\begin{bmatrix} 0 & 0 & 11 & 0 \\ 12 & 0 & 0 & 0 \\ 0 & -4 & 0 & 0 \\ 0 & 0 & 0 & -15 \end{bmatrix}$$

Fig 3.2: Sample Sparse Matrix

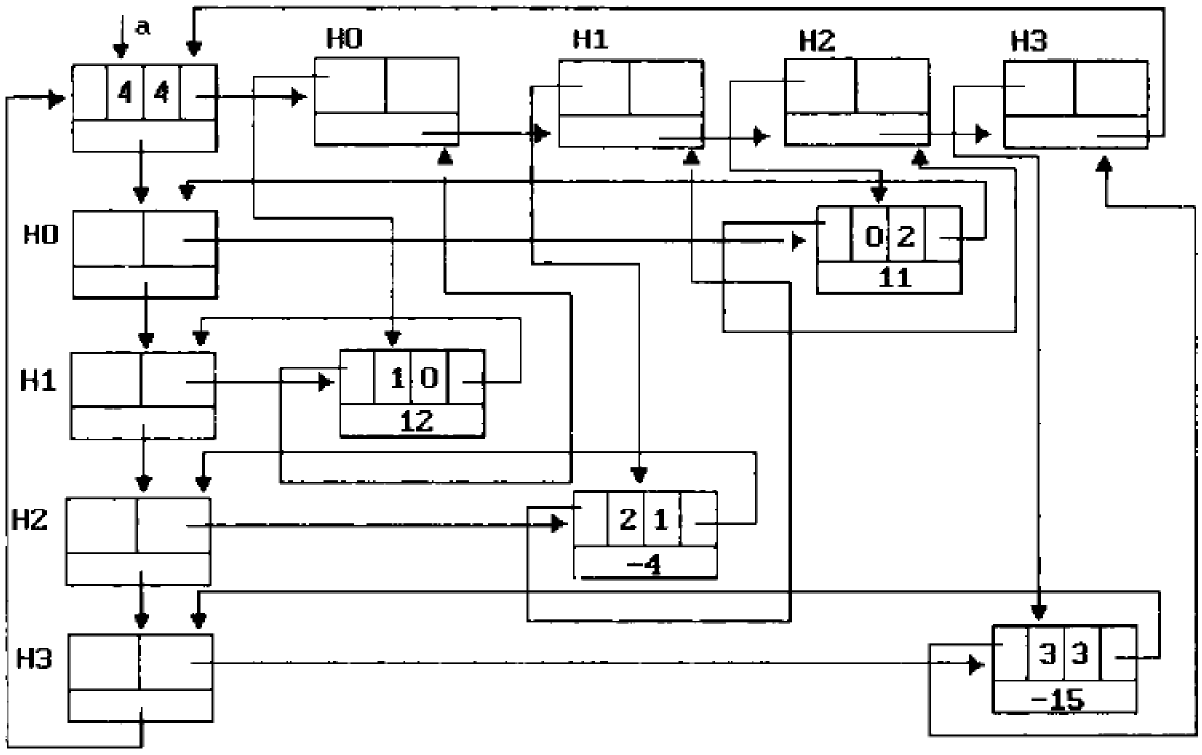


Fig 3.3: Linked representation of this matrix

If we wish to represent a num-rows x num-cols matrix with num-terms nonzero terms, then we need max {num-rows, num-cols] + num-terms + 1 nodes. While each node may require several words of memory, the total storage will be less than num-rows * num-cols when num-terms is sufficiently small.

```
#define MAX-SIZE 50 /*size of largest matrix*/
typedef enum {head,entry} tagfield;
typedef struct matrix—node *matrix_pointer;
typedef struct entry—node
{
     int row;
     int col;
     int value;
     } ;
     typedef struct matrix—node
     {
     matrix—pointer down;
     matrix—pointer right;
     tagfield tag;
     union {
     matrix—pointer next;
     entry—node entry;
```

```
    } u;
} ;
matrix-pointer hdnode[MAX—SIZE];
```

3.2 DOUBLY LINKED LIST

1. The difficulties with single linked lists is 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 linkedlist. 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 is divided 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 in the list

Example Doubly linked list representation is shown in Fig 3.4.

The declarations are:

```
typedef struct node *nodePointer;

typedef struct {

            nodePointer llink;
            element data;
            nodePointer rlink;

        } node;
```

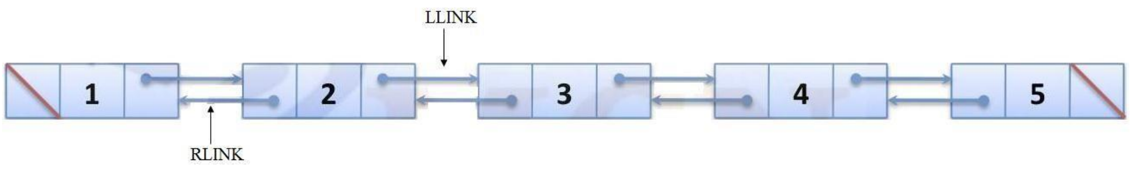


Fig 3.4: Doubly linked list representation

Insertion into a doubly linked list

Insertion into a doubly linked list is fairly easy. Assume there are two nodes, node and newnode, 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;
          }
```

Program: 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 deleted from 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);
                              }
                              }
```

Program: Deletion from a doubly linked circular list

3.3 Circular Linked Lists

A circular linked list is a linked list in which the last node points to the head or front node making the data structure to look like a circle. A circularly linked list node can be implemented using singly linked or doubly linked list.

The below representation shows how a circular linked list looks like. Unlike the linear linkedlist, the last node is pointed to the head using a rear pointer. A circular linked list is a linked list in which the last node points to the head or front node making the data structure to look like a circle. A circularly linked list node can be implemented using singly linked or doublylinked list.

The below representation shows how a circular linked list looks like. Unlike the linear linkedlist, the last node is pointed to the head using a rear pointer as shown in Fig 3.5.

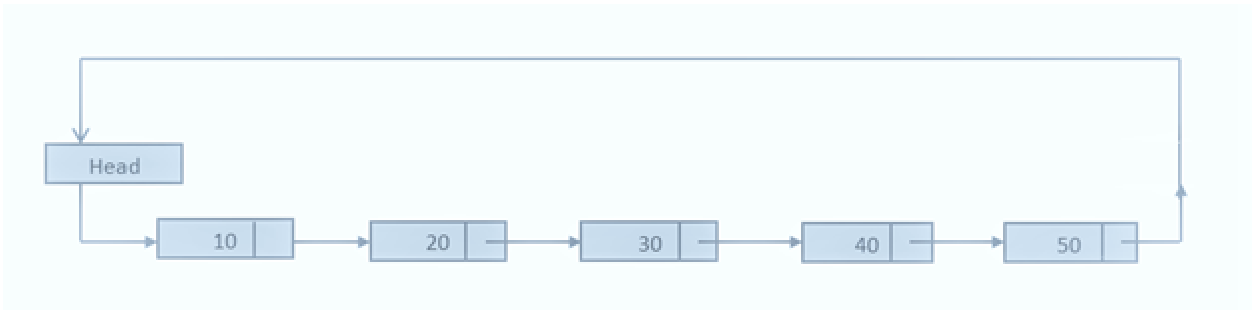


Fig 3.5: Circular Linked list representation

Program for Circular Linked List in C

We basically have 3 nodes head, rear and temp to implement a circular linked list. The rear points to the last node of the list, head points to first node. This helps to maintain a track on where is the front and rear nodes of the circle. Let us see the basic operations such as creation, deletion and displaying of the elements of the circular linked list.

Creation of Node

This is much similar to that of creating a node in singly linked list. It involves creating a newnode and assigning data and pointing the current node to the head of the circular linked list. The code is as shown below.

```
void create ()
{
        node *newnode;
        newnode=(node*) malloc(sizeof(node));
        printf("\nEnter the node value : ");
        scanf("%d”, &newnode->info);
        newnode->next=NULL;
        if(rear==NULL)
        front=rear=newnode;
        else
        {
                rear->next=newnode;
                rear=newnode;
        }

        rear->next=front;
}
```

Deletion of Node

We conventionally delete the front node from the list in this program. To delete a node, we need to check if the list is empty. If it is not empty then point the rear node to the front->next and rear->next to front. This removes the first node.

```
void del()
{
        temp=front;
        if(front==NULL)
                printf("\nUnderflow :");
        else
        {
                if(front==rear)
                {
                        printf("\n%d”, front->info);
                        front=rear=NULL;
                }
                else
                {
                        printf("\n%d”, front->info);
                        front=front->next;
                        rear->next=front;
                }

        temp->next=NULL;
        free(temp);
        }
}
```

Traversing Circular Linked List

Traversing the circular list starts from front node and iteratively continues until the rear node. The following function is used for this purpose.

```
void display ()
{
        temp=front;
        if(front==NULL)
                printf("\nEmpty");
        else
        {
                printf("\n");
                for (; temp! =rear;temp=temp->next)
                        printf("\n%d address=%u next=%u\t",temp->info,temp,temp->next);
                        printf("\n%d address=%u next=%u\t",temp->info,temp,temp->next);
        }
}
```

Complete Program

```
#include<stdio.h>
#include<stdlib.h>

typedef struct Node

{
        int info;
        struct Node *next;
}node;

node *front=NULL,*rear=NULL,*temp;

void create();
void del();
void display();

int main()
{
        int chc;
        do
        {
        printf("\nMenu\n\t 1 to create the element : ");
        printf("\n\t 2 to delete the element : ");
        printf("\n\t 3 to display the queue : ");
        printf("\n\t 4 to exit from main : ");
        printf("\nEnter your choice : ");
        scanf("%d",&chc);

                switch(chc)
                {
                        case 1:
                        create();
                        break;

                        case 2:
                        del();
                        break;

                        case 3:
                        display();
                        break;

                        case 4:
                        return 1;

                        default:
                                printf("\nInvalid choice :");
                }
        }while(1);

        return 0;
}

void create()
{
        node *newnode;
        newnode=(node*)malloc(sizeof(node));
        printf("\nEnter the node value : ");
        scanf("%d",&newnode->info);
        newnode->next=NULL;
        if(rear==NULL)
        front=rear=newnode;
        else
        {
                rear->next=newnode;
                rear=newnode;
        }

        rear->next=front;
}

void del()
{
        temp=front;
        if(front==NULL)
                printf("\nUnderflow :");
        else
        {
                if(front==rear)
                {
                        printf("\n%d",front->info);
                        front=rear=NULL;
                }
                else
                {
                        printf("\n%d",front->info);
                        front=front->next;
                        rear->next=front;
                }
```

```
            temp->next=NULL;
            free(temp);
            }
    }

    void display()
    {
            temp=front;
            if(front==NULL)
                    printf("\nEmpty");
        else
        {




        }
}
```

3.4 TREES

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. An example tree is shown in Fig 3.6.

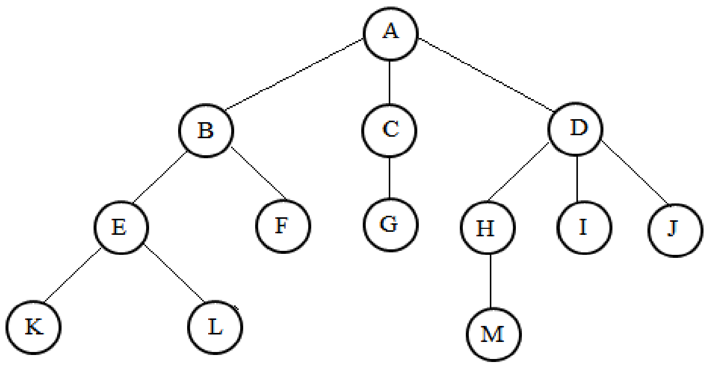


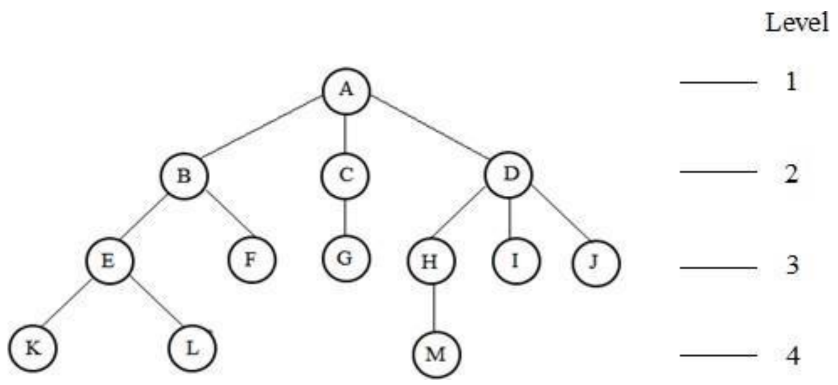
Fig 3.6: Tree representation

Every node in the tree is the root of some subtree

Terminologies

- ***Node:*** The item of information plus the branches to other nodes
- ***Degree:*** The number of subtrees of a node
- ***Degree of a tree:*** The maximum of the degree of the nodes in the tree.
- ***Terminal nodes (or leaf):*** nodes that have degree zero or node with no successor
- ***Nonterminal nodes***: nodes that don't belong to terminal nodes.
- ***Parent and Children:*** Suppose N is a node in T with left successor S1 and right successor S2, then N is called the Parent (or father) of S1 and S2. Here, S1 is called left child (or Son) and S2 is called right child (or Son) of N.
- ***Siblings:*** Children of the same parent are said to be siblings.
- ***Edge:*** A line drawn from node N of a T to a successor is called an edge
- ***Path:*** A sequence of consecutive edges from node N to a node M is called a path.
- ***Ancestors of a node:*** All the nodes along the path from the root to that node.
- ***The level of a node:*** defined by letting the root be at level zero. If a node is at level *l*,then it children are at level *l+1*.
- ***Height (or depth):*** The maximum level of any node in the tree

Example



A is the root node
B is the parent of E and F
C and D are the sibling of B E and F are the children of B
K, L, F, G, M, I, J are external nodes, or leaves A, B, C, D, E,
H are internal nodes
The level of E is 3
The height (depth) of the tree is 4 The degree of node B is 2
The degree of the tree is 3
The ancestors of node M is A, D, H
The descendants of node D is H, I, J, M

Representation of Trees

There are several ways to represent a given tree such as

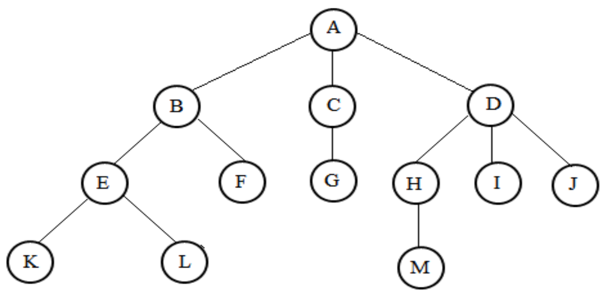


Fig 3.7: Tree example

1. List Representation
2. Left Child- Right Sibling Representation
3. Representation as a Degree-Two tree

List Representation:

The tree can be represented as a List. The tree of Fig 3.7 could be written as the list.

(A (B (E (K, L), F), C (G), D (H (M), I, J)))

- The information in the root node comes first.

- The root node is followed by a list of the subtrees of that node.

Tree node is represented by a memory node that has fields for the data and pointers to the tree node's children. List representation of tree is shown in Fig 3.8.

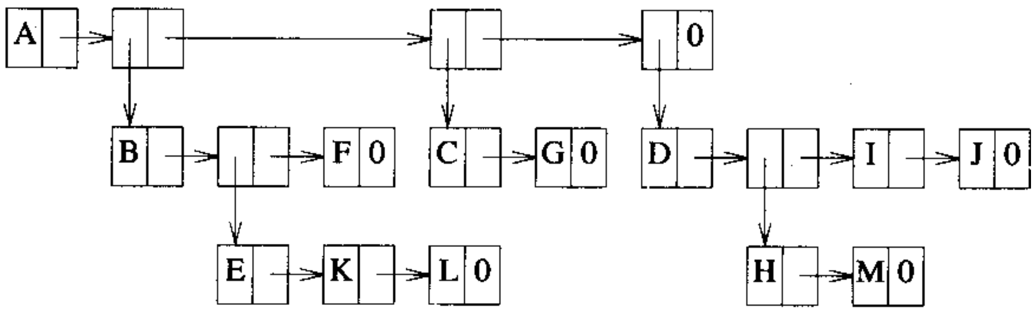
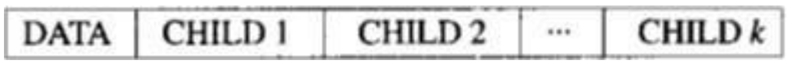


Fig 3.8: List representation of a tree

Since the degree of each tree node may be different, so memory nodes with a varying number of pointer fields are used.

For a tree of degree k, the node structure can be represented as below figure. Each child field is used to point to a subtree.



DATA	CHILD 1	CHILD 2	···	CHILD k

Left Child-Right Sibling Representation

The below Fig 3.9 shows the node structure used in the left child-right sibling representation.

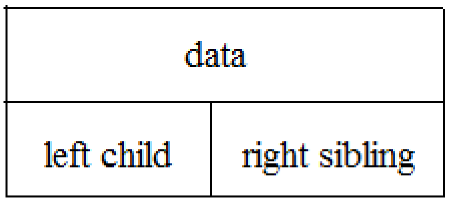


Fig 3.9: Left child right sibling node structure

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.

Ex:

- In Fig 3.7, the leftmost child of A is B, and the leftmost child of Dis 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).

Fig 3.10 shows the tree of Figure (A) redrawn using the left child-right sibling representation.

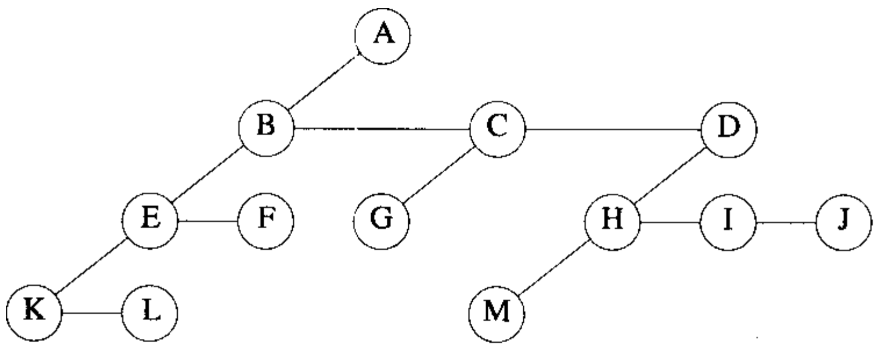


Fig 3.10: Left child-right sibling representation of tree of Fig 3.7

Representation as a Degree-Two 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 Fig 3.11.

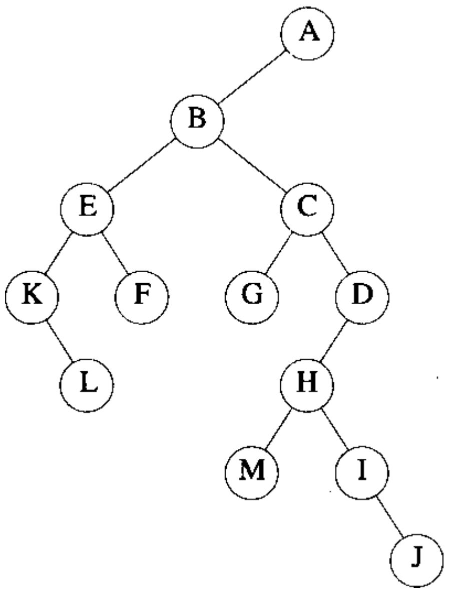


Fig 3.11: degree-two representation

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

3.5 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 as shown in Fig 3.12.

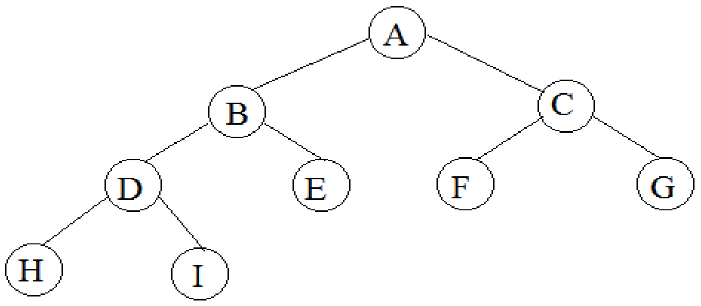


Fig 3.12: Binary Tree

Different kinds of Binary Tree

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 as shown in Fig 3.13a.

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

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 farleft as possible as shown in Fig 3.13b.

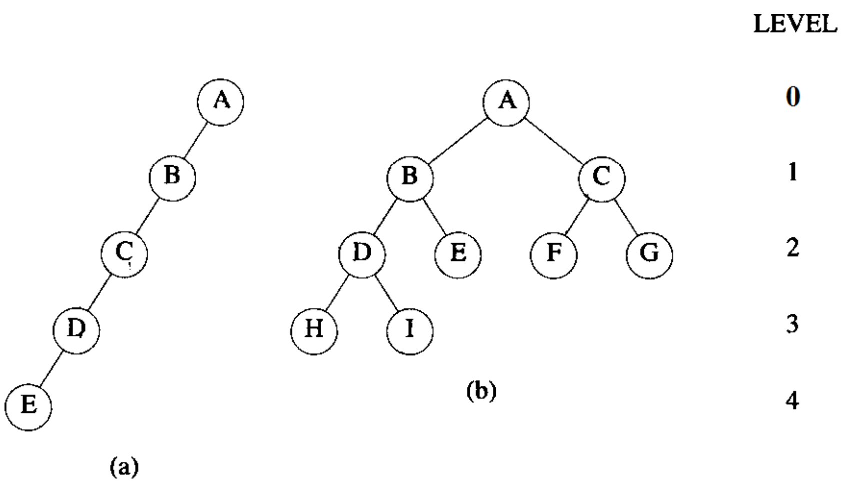


Fig 3.13 (a): Skewed binary tree (b): Complete binary tree

Full Binary Tree

A full binary tree of depth ‘k’ is a binary tree of depth k having $2^k - 1$ node, $k \geq 1$ as shown in Fig 3.14.

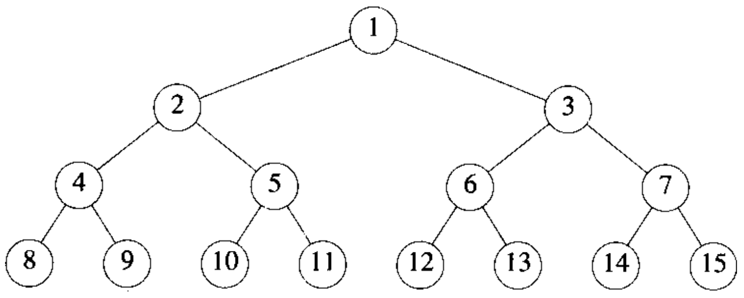


Fig 3.14: Full binary tree of level 4 with sequential node number

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 shown in Fig 3.15.

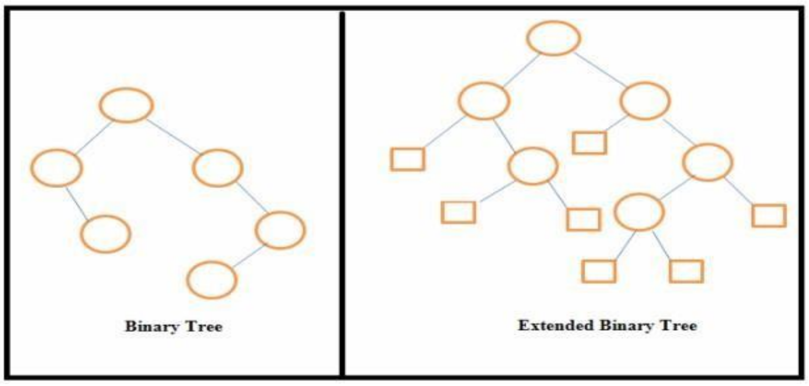


Fig 3.15: Binary Tree and its extended binary tree

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 numberof 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 thatthe 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 k k

$\sum$ (maximum number of nodes on level i) = $\sum 2^{i-1}$

= 2^{k-1} i=0 i=0

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

For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number ofnodes of degree 2, then $n_0 = n_2 + 1$.

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

$= n_0 + n_1 + n_2$ (1)

Count the number of branches in a binary tree. If B is the number ofbranches, 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= $n_1 + 2n_2 + 1$ (2)

Subtracting Eq. (2) from Eq. (1) and rearranging terms, we get $n_0 = n_2 + 1$

Consider the Fig 3.16:

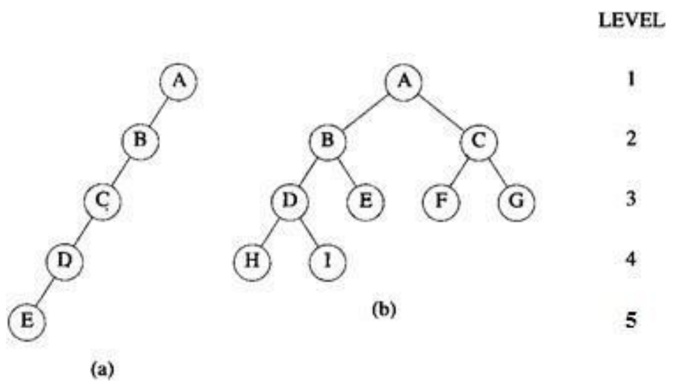


Fig 3.16: Example binary tree

Here, For Fig 3.16 (b) $n_2=4$, $n_0= n_2+1=4+1=5$

Therefore, the total number of leaf node=5

3.6 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 Fig 3.18 shows the array representation for both the trees of Fig 3.17.

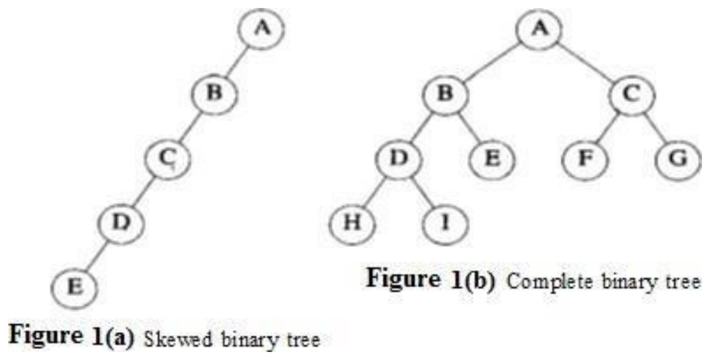
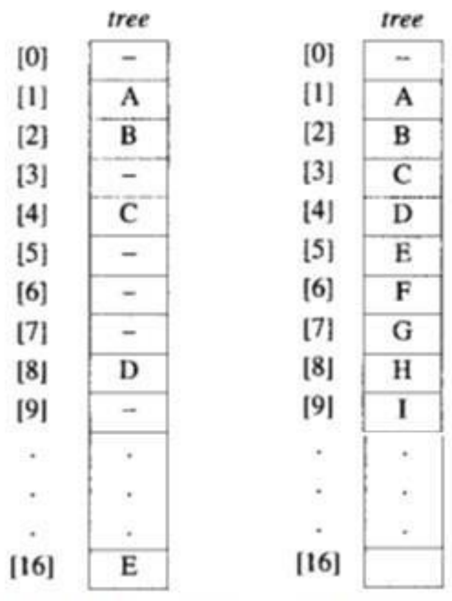

A
B
C
D
E
Figure 1(a) Skewed binary tree
A
B
C
D
E
F
G
H
I
Figure 1(b) Complete binary tree

Fig 3.17: Skewed and Complete binary tree



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

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

Fig 3.18: Array representation of binary trees in Fig 3.17

- 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;
```

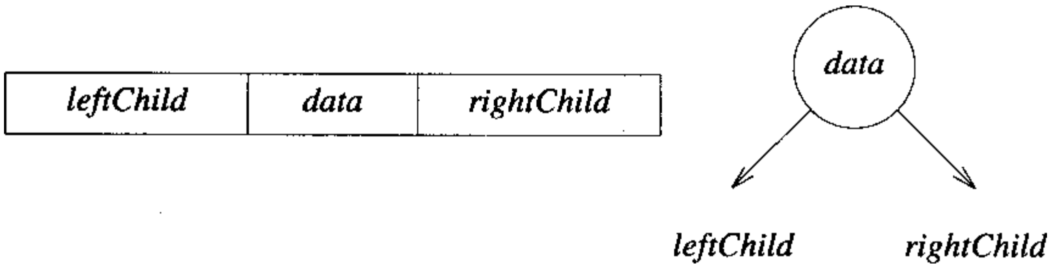


Fig 3.19 : Node representation

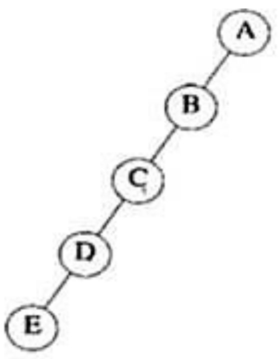


Figure 1(a) Skewed binary tree

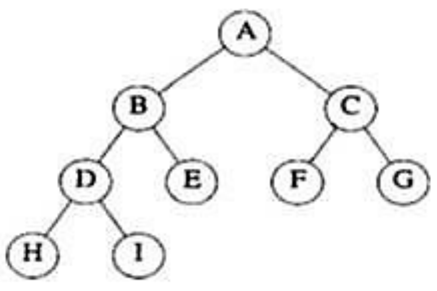


Figure 1(b) Complete binary tree

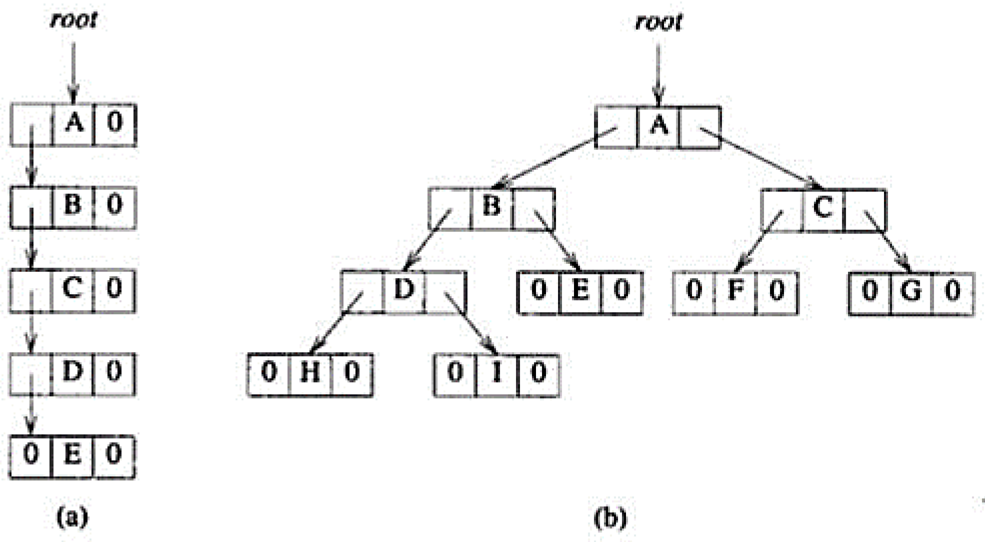


Fig 3.20: Linked representation of the binary tree

3.7 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

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 andresume.

Recursion function:

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

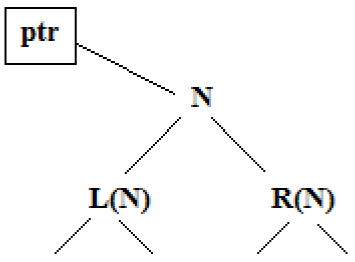
- Visit the root
- Traverse the left subtree in preorder.
- Traverse the right subtree in preorder

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

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

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

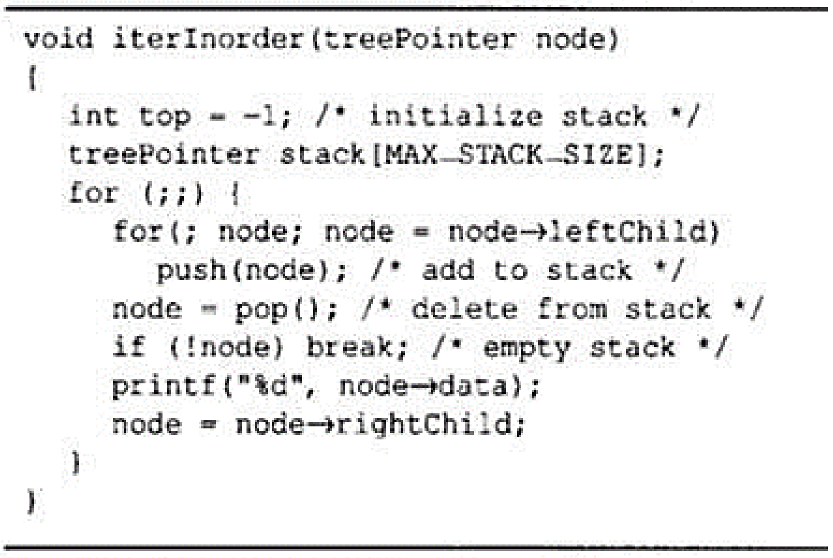
- Traverse the left subtree in postorder.
- Traverse the right subtree in postorder.
- Visit the root

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

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 nullnode is reached. The traversal then continues with the left child. The traversal is completewhen 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;
   }
}
```

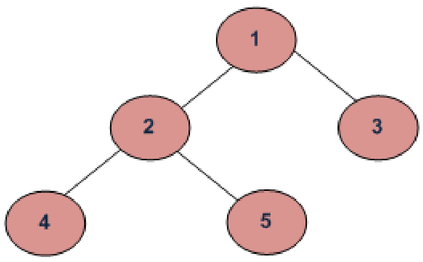
Program : Iterative inorder traversal

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.

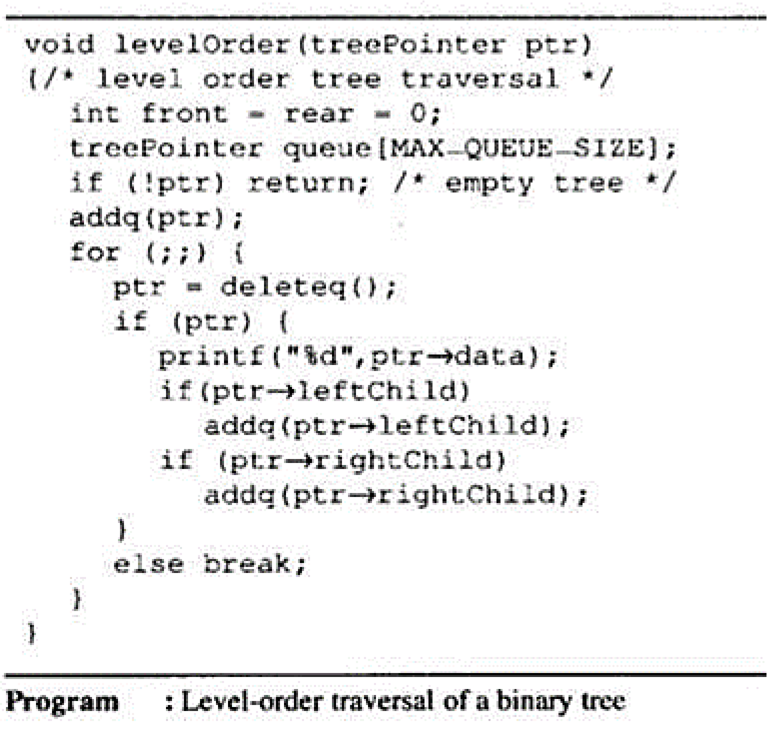
Firstly, visit the root, then the root's left child, followed by the root's right child. Thus continuing in this manner, visiting the nodes at each new level from the leftmost node to the rightmost node.



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

3.8 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 →left Child is null, then replace the null link with a pointer to the inorder predecessor of ptr.
2. If ptr →right Child is null, replace the null link with a pointer to the inorder successor of ptr.

Ex: Consider the binary tree as shown in below in Fig 3.21:

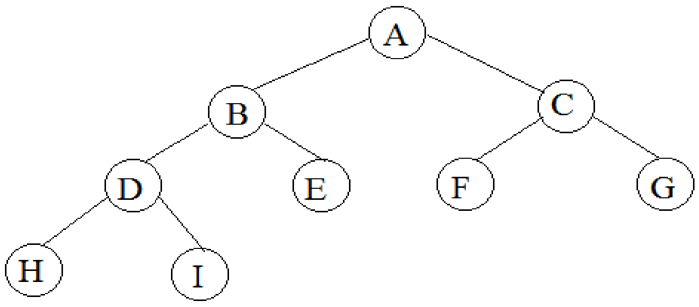


Fig 3.21: Binary Tree

There should be no loose threads in threaded binary tree. But in Fig 3.22 two threads have been left dangling: one in the left child of *H*, the other in the right child of G.

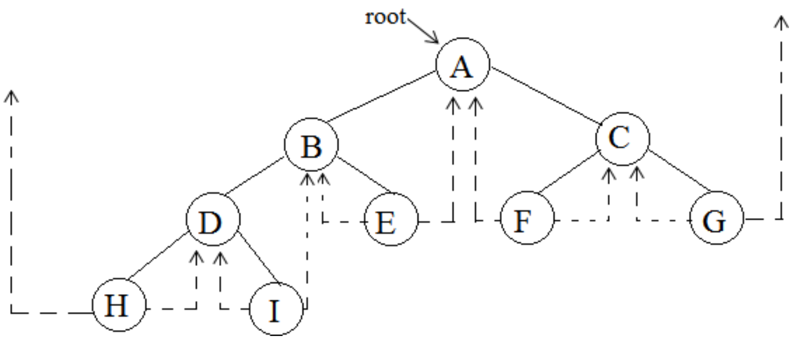


Fig 3.22: Threaded tree corresponding to Fig 3.21

In above figure the new threads are drawn in broken lines. This tree has 9 nodes 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., left Thread and *right Thread*

- 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;
```

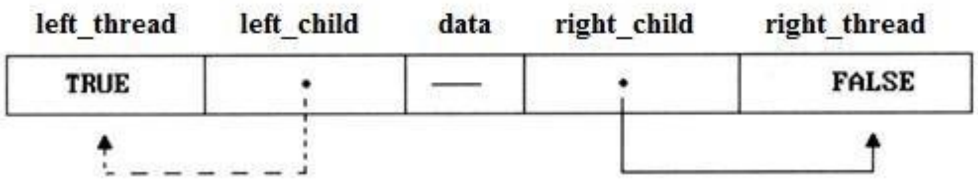


Figure An empty threaded tree

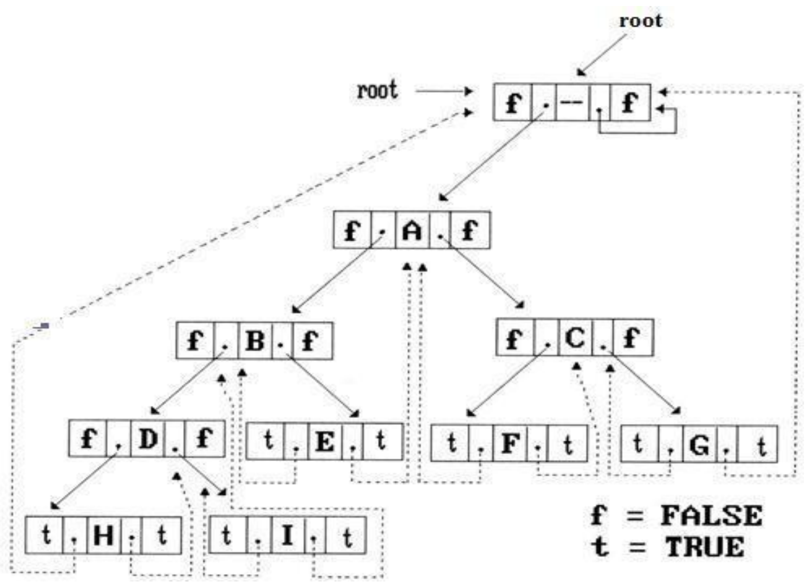


Fig 3.23: Memory representation of tree

The complete memory representation for the tree of figure is shown in Fig 3.23.

The variable ***root*** points to the header node of the tree, while root →**leftChild** points tothe 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 inorder (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 Fig 3.24
- 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.

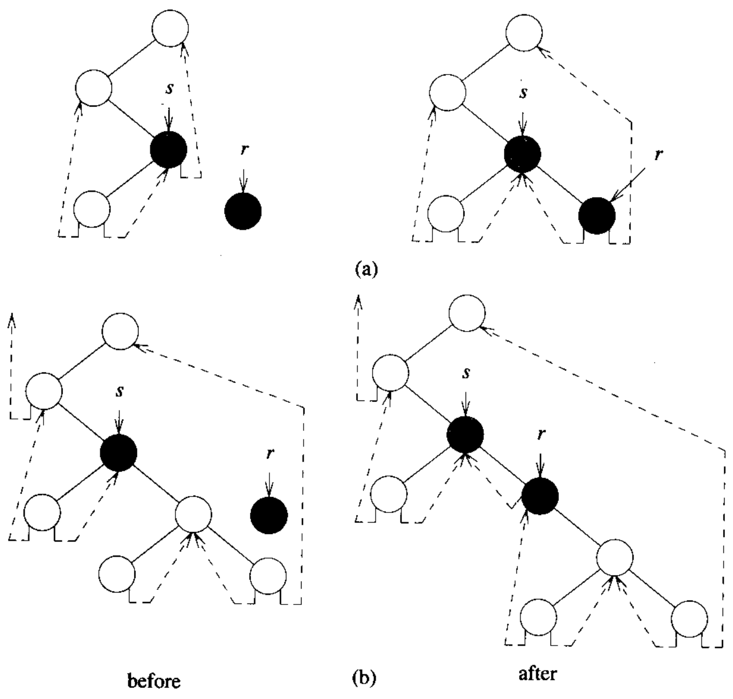


Fig 3.24: Inserting a node in threaded binary tree

```
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;
    }
    }
```