the operator.

8.14 HUFFMAN TREE

As we have seen earlier that an extended Binary tree is a binary tree in which every node has zero or two children. The nodes which have two children are called internal nodes and which have no children are called external nodes.

In any 2 Tree the number of external nodes is 1 more than the number of internal nodes *i.e.*,

$$\boxed{E = I + 1}$$

Wher E is the number of external nodes and I is the number of internal nodes.

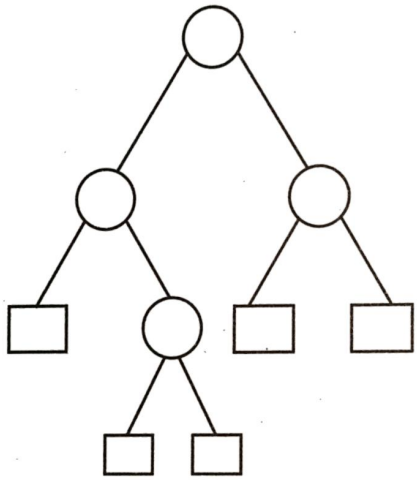


Fig. 8.55.

Internal nodes = 4

External nodes = 5

We know, the **path length** for any node is the number of minimum nodes traversed from root to that node. With this in mind, we define the External path length. L_E of a 2 Tree to be the sum of all path lengths *i.e.*,

$$L_E = 2 + 3 + 3 + 2 + 2 = 12$$

Similarly total path length for internal nodes are

$$L_I = 0 + 1 + 2 + 1 = 4$$

We can also get the total path length of external node through

$$\boxed{L_E = L_I + 2n}$$

Where n is the total number of nodes in the tree.

Here,
$$L_E = 4 + 2 * 4$$
$$= 4 + 8 = 12$$

Suppose every external node has same weight W, then the weighted path length for the external node will be

$$P = W_1L_1 + W_2L_2 + W_3L_3 + ... + W_nL_n$$

Where W_i and L_i denote respectively, the weight and path length of an external node.

Suppose we create different trees which have same weigths on external nodes then it is not necessary that they have same weighted path length.

Let us take the weights 2, 7, 9, 10 and create three different trees.

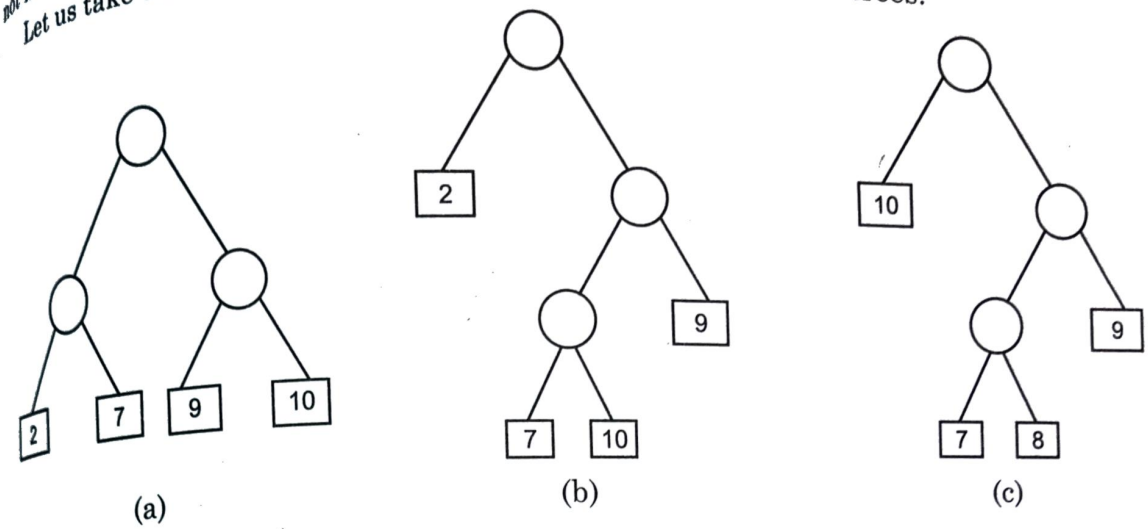


Fig. 8.56.

$$\begin{cases} P_1 = 2*2+7*2+9*2+10*2 = 4+14+18+20 = 56 \\ P_2 = 2*1+7*3+10*3+9*2 = 2+21+30+18 = 71 \\ P_3 = 10*1+7*3+8*3+9*2 = 10+21+24+18 = 73 \end{cases}$$

We can see that three different trees have different path lengths even same tree of (*b*) and (*c*) have different path lengths. So the problem arises for obtaining a unique tree which has minimum path length.

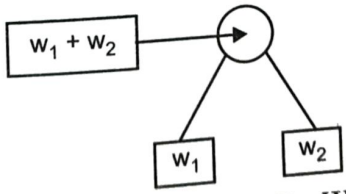
The general problem that we want to solve is as follows. Suppose a list of n weights is given:

$$W_1, W_2, \ldots\ldots, W_n$$

Among all the 2-trees with n external nodes and with the given n weights, find a tree T with a minimum weighted path length. Huffman gave an algorithm to find such a tree.

Huffman Algorithm:

1. Suppose, there are n weights $W_1, W_2, \ldots\ldots, W_n$.
2. Take two minimum weights among the n given weights. Suppose W_1 and W_2 are first two minimum weights then sub-tree will be:



3. Now the remaining weights will be $W_1 + W_2, W_3, W_4, \ldots\ldots W_n$.
4. Create all sub-tree at the last weight.

Thus, Huffman algorithm constructing the tree from the bottom up rather than from the top down approach.

EXAMPLE 8.11. Suppose *A, B, C, D, E, F* and *G* are 7 elements with weights as follows:

Item	A	B	C	D	E	F	G
Weights	15	10	5	3	7	12	25

Create an extended binary tree by Huffman algorithm.

Solution: Step 1: Taking two nodes with minimum weights *i.e.,* 3 and 5.

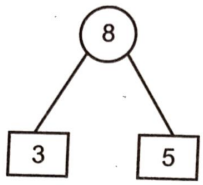


Fig. 8.57.

Now elements in the list are:

15 10 8 7 12 25

Step 2: Taking two minimum weight nodes which are 8 and 7 *i.e.,* now.

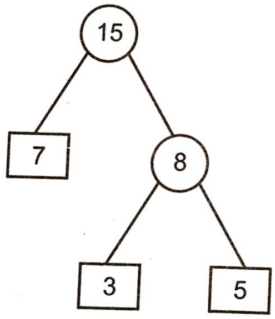


Fig. 8.58.

Now elements in the list are:

15 10 15 12 25

i.e.,

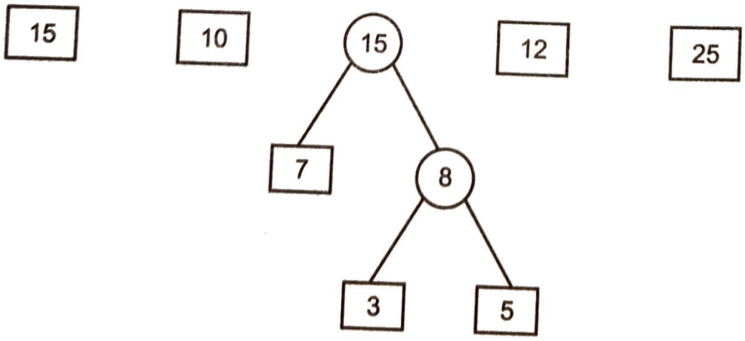


Fig. 8.59.

Step 3: Taking two minimum weighted nodes *i.e.,* 10 and 12.

Now,

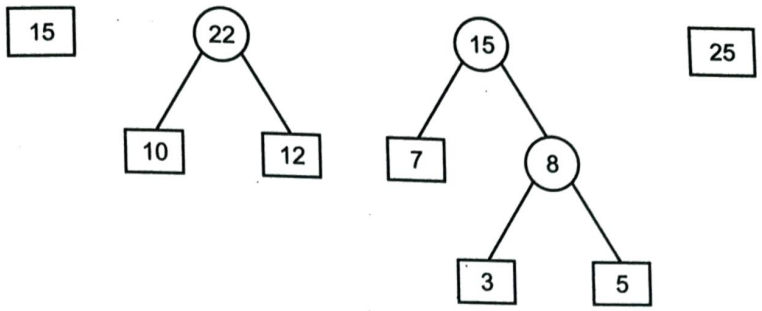


Fig. 8.60.

i.e., elements in the list are
15, 22, 15, 25.

Step 4: Taking two minimum weighted nodes i.e., 15 and 15.

Now,

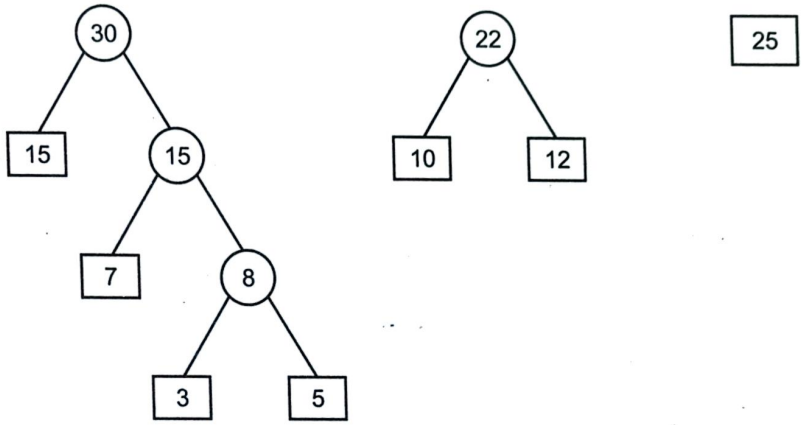


Fig. 8.61.

i.e., elements in the list are
30, 22, 25.

Step 5: Taking two nodes with minimum weights which are 22 and 25.

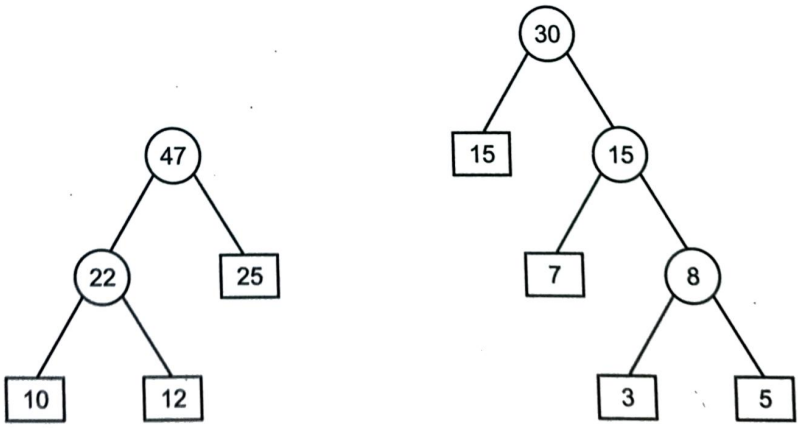


Fig. 8.62.

Now the elements in the list are 47 and 30

Step 6: Taking two nodes with minimum weights which are 30 and 47.

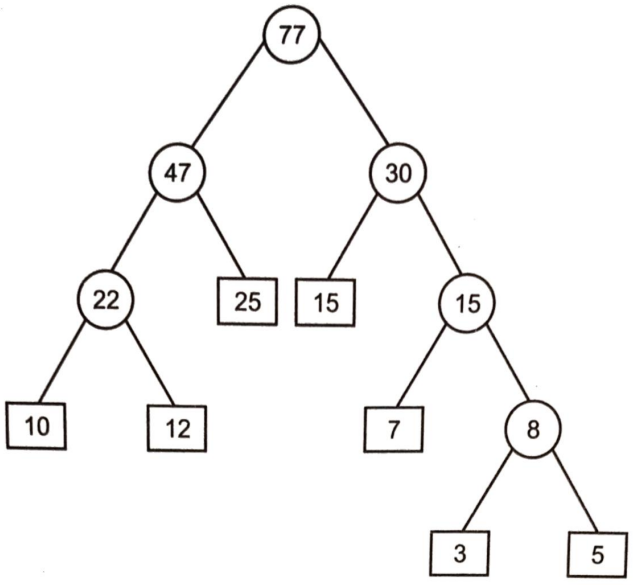


Fig. 8.63 Huffman Tree

8.14.1 Application of Huffman Algorithm

The Huffman algorithm is used to perform the encoding and decoding of a long message consisting of a set of symbols. Suppose we want to send a message there are two options either it sends data as fixed size or to send it as variable length size.

Suppose a collection of n data items A_1, A_2, ..., A_n; are to be coded by means of strings of bits. One way to do this is to code each item by an r-bit string where,

$$2^{n-1} < n \leq 2^r$$

E.g., a 48-character set is frequently coded in memory by using 6-bit strings. One cannot use 5-bit strings since $2^5 < 48 < 2^6$.

Suppose the data items do not occur with the same probability. Then memory space may be conserved by using variable-length strings, were items which occur frequently an assigned shorter strings and items which occur infrequently are assigned longer strings.

Now, we discuss a coding technique using variable-length string that is based on the Huffman Tree for weighted data item.

A **Huffman tree** has to be constructed to perform both encoding and decoding. The Huffman tree has to be constructed to perform both encoding and decoding. The Huffman tree is constructed based on the frequency of the symbols of a given message. The main aim of using the Huffman algorithm is to minimise the length of the encoded message by assigning a shorter bit string to the frequently occurring symbols in the message.

Algorithm

1. Accept the total number of symbols, say *n*.
2. Accept the *n* different symbols and their frequencies.
3. Accept the message to be encoded.
4. Find the two symbols from the given set of symbols such that they occur less frequently.

5. Assign codes (it can be n bits where n can be as minimum as possible) to these chosen symbols such that the codes differ in their last bit positions. One symbol will have 0 as the last bit, the other symbol will have 1 as the last bit.
6. Combine these two chosen symbols by forming a parent node from the nodes corresponds to the chosen symbols. The frequency of the parent node is the sum of the frequencies of the nodes corresponding to the chosen symbols. Now this formed parent node can be treated as a symbol but it will be an internal node.
7. Now choose the second symbol such that its frequency is minimum.
8. Repeat steps 5, 6 and 7 as long as there are some symbols for that node that are not formed.

Once the Huffman tree is formed, the code of any symbol can be formed by starting at that node and climbing up to the root node. Initially the code for that symbol is NULL. If a left branch is climbed, 0 is appended at the beginning of the code. If a right branch is climbed, 1 is appended at the beginning of the code.

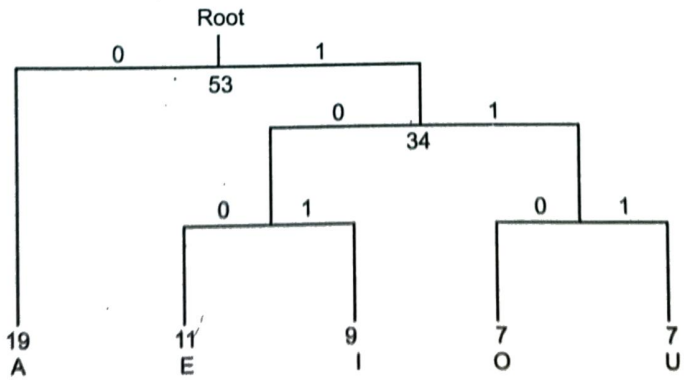
Huffman Coding

Huffman follows a bottom-up approach. Procedure which we adopted is as follows :

1. Form the frequency list of all the symbols in the descending order.
2. Locate the two symbols in the list with the lowest frequencies. Frequencies referred as weights.
3. Create a parent node for these two nodes whose weight is equal to the sum of weights of two child nodes.
4. Remove the two child from the list and add newly created parent node to the list along-with the weight.
5. Repeat thorugh the step (2) until only one node is left in the tree.

EXAMPLE:

Symbol	Frequency
A	19
E	11
I	9
O	7
U	7



Thus, by using this technique, the codes for all the symbols are given in the following table.

Symbol	Codes
A	0
E	100
I	101
O	110
U	111

To perform the decoding operation, just traverse the nodes from the root node towards the leaf nodes of the Huffman tree such that the given code bits are traversed. The important features of the Huffman tree are:

1. The Huffman tree is a binary tree.
2. In the Huffman tree, the most frequently occuring symbols will be the leaf nodes near the root node whereas the least frequently occuring symbols will be farther away from the root node.
3. The most frequently occuring symbols will have smaller code bits whereas, the least frequently occuring symbols will have more code bits.
4. The root node of the Huffman tree is not assigned any code. The left child of the root node is assigned the code 0. The right child of the root node is assigned the node 1.
5. All the symbols of the given message will be leaf nodes of the Huffman tree.
6. The code for the child nodes of the same parent differ only in the last bit. The last-bit is 0 for the left-child node and is 1 for the right child node.

The Huffman tree finds applications wherever the messages have to be transmitted and received with less number of code bits.

8.15 THREADED BINARY TREES

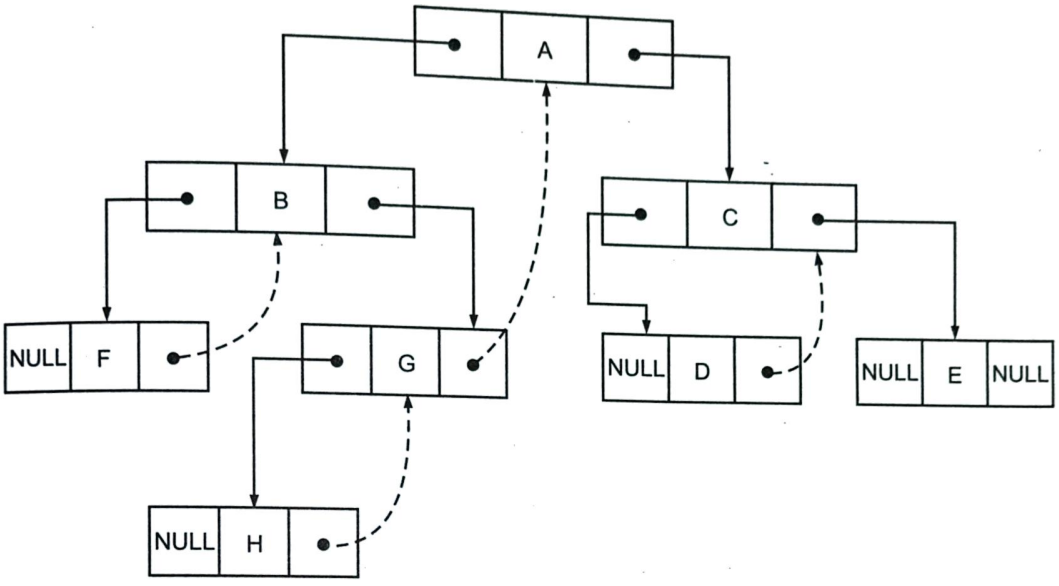
Consider the linked representation of any binary tree, we notice that, half of the entries in the pointer fields LEFT and RIGHT will contain null entries. This space may be more efficiently used by replacing the null entries by special pointers, called **Threads,** which point to the nodes higher in the trees. Such trees are called **Threaded Trees.**

Note: A binary tree with n nodes, when $n \geq 0$ has exactly $n + 1$ null links.

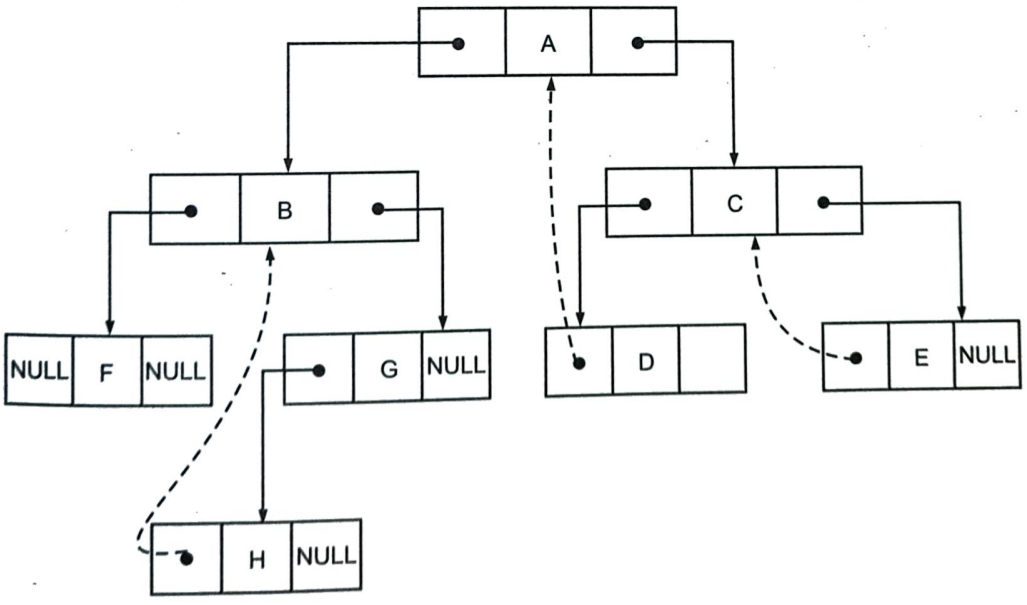
Threads in a binary tree must be distinguished from normal pointers. In the graphical representation of a threaded binary tree, the threads are shown by dotted lines. In computer memory, an extra field, called tag or flag is used to distinguish a thread from a normal pointer. Trees can be threaded using one-way threading or two-way threading. In a one-way threading a therad will appear in the **right field** of the node and will point to the successor node in the inorder traversal of tree; and in two-way threading of tree a thread will also appear in the **left field** of a node and will point to the preceding node in the inorder traversal of tree. So, there are many ways to thread a bianry tree. These are:

1. The right NULL pointer of each node can be replaced by a thread to the successor of that node under in-order traversal called a **right thread,** and the tree will called a **right threaded tree.**
2. The left NULL pointer of each node can be replaced by a thread to the predecessor of that node under in-order traversal called a left-thread and the tree will called a **left-threaded tree.**

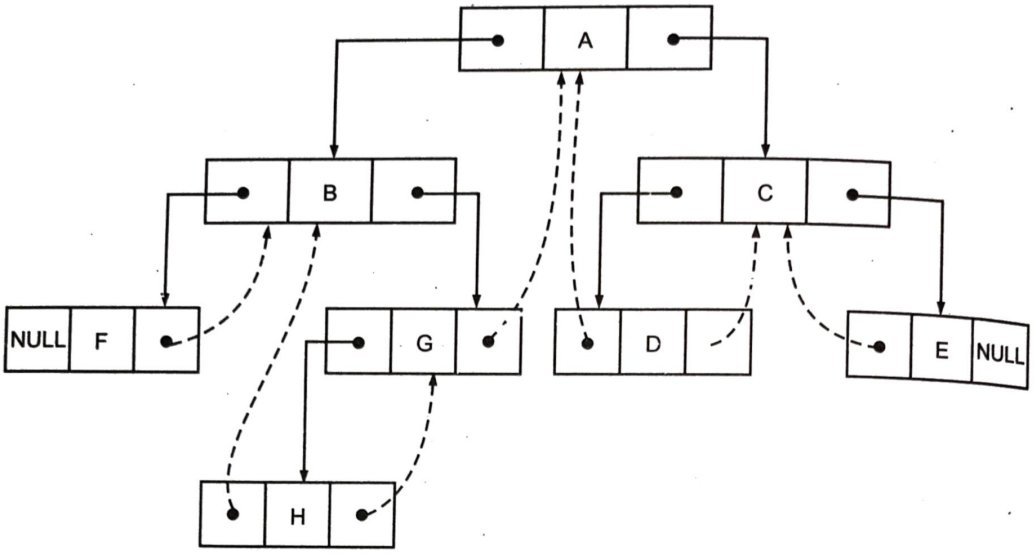
3. Both left and right NULL pointers can be used to point to predecessor and successor of that node, respectively, under in-order traversal. Such a tree is called a **fully threaded tree.**



(a) Right threaded binary tree



(b) Left threaded binary tree



(c) Fully threaded binary tree

Fig. 8.64.

To implement a **right threaded binary tree** an one extra field **rthread** is used. If rthread is assigned to 1 then its corresponding right link represents a thread and if rthread is assigned to 0 then right link represents an ordinary link connecting to the right sub-tree. The *C* node structure can be defined as:

```
struct node
    {
        char info;
        struct node * left;
        struct node * right;
        int rthread;  /*1 indicate threaded link
                        0 indicate ordinary link */
    };
typedef struct node * NODE;
```

Similarly to implement a **left threaded binary tree,** we need one extra field **lthread**. It is assigned to 1 indicate the presence of thread and 0 indicate an ordinary link connecting to the left sub-tree. The *C* node structure can be defined as:

```
struct node
    {
        char info;
        struct node * left;
        struct node * right;
        int lthread;
    }
typedef struct node * NODE;
```

To implement **fully threaded binary tree,** we need two extra field of lthread and rthread as defined earlier are used and the C node structure can be defind as:

```
struct node
    {
        char info;
        struct node * left;
        struct node * right;
        int lthread;
        int rthread;
    };
typedef struct node * NODE;
```

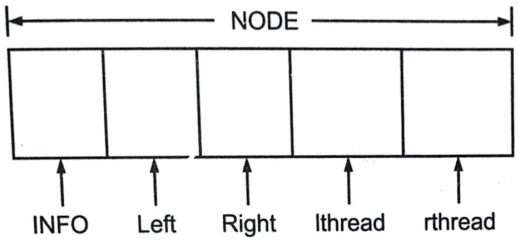


Fig. 8.65 Node of a threaded binary tree

Now, we have seen that if left or right pointer of a node is NULL then it is made a thread pointing to the inorder predecessor or inorder successor of the node. First node in inorder traversal has no predecessor and last node has no successor, so the left pointer of the left most node which is the first node in order traversal and the right pointer of the right most node which is the last node in inorder traversal contains NULL. So we can take a dummy node called the **header node** and we will represent are tree as the left sub-tree of this header node. Left pointer of this header node will point to the root node of our tree. When our tree will be empty then left pointer of this header node will be a thread ponting to itself.

So, now the left most node and right most node of our two-way threaded binary tree will not contain NULL, they will contain threads pointing to this header node.

The structure of a node in a threaded binary tree will be as:

```
typedef enum{thread, link}boolean;
struct node
    {
        struct node * leftptr;
        boolean left;
        int info;
    struct node * rightptr;
        boolean right;
    };
```

Here we have taken two boolean members left and right to differentiate between a thread and link. These members can take values thread or link.

If left = link pointer leftptr points to the left child of the node.
If left = thread pointer leftptr is a thread pointing to inorder predecessor of the node.
If right = link pointer rightptr points to the rightchild of the node.
If right = thread pointer rightptr is a thread pointing to inorder successor of the node.

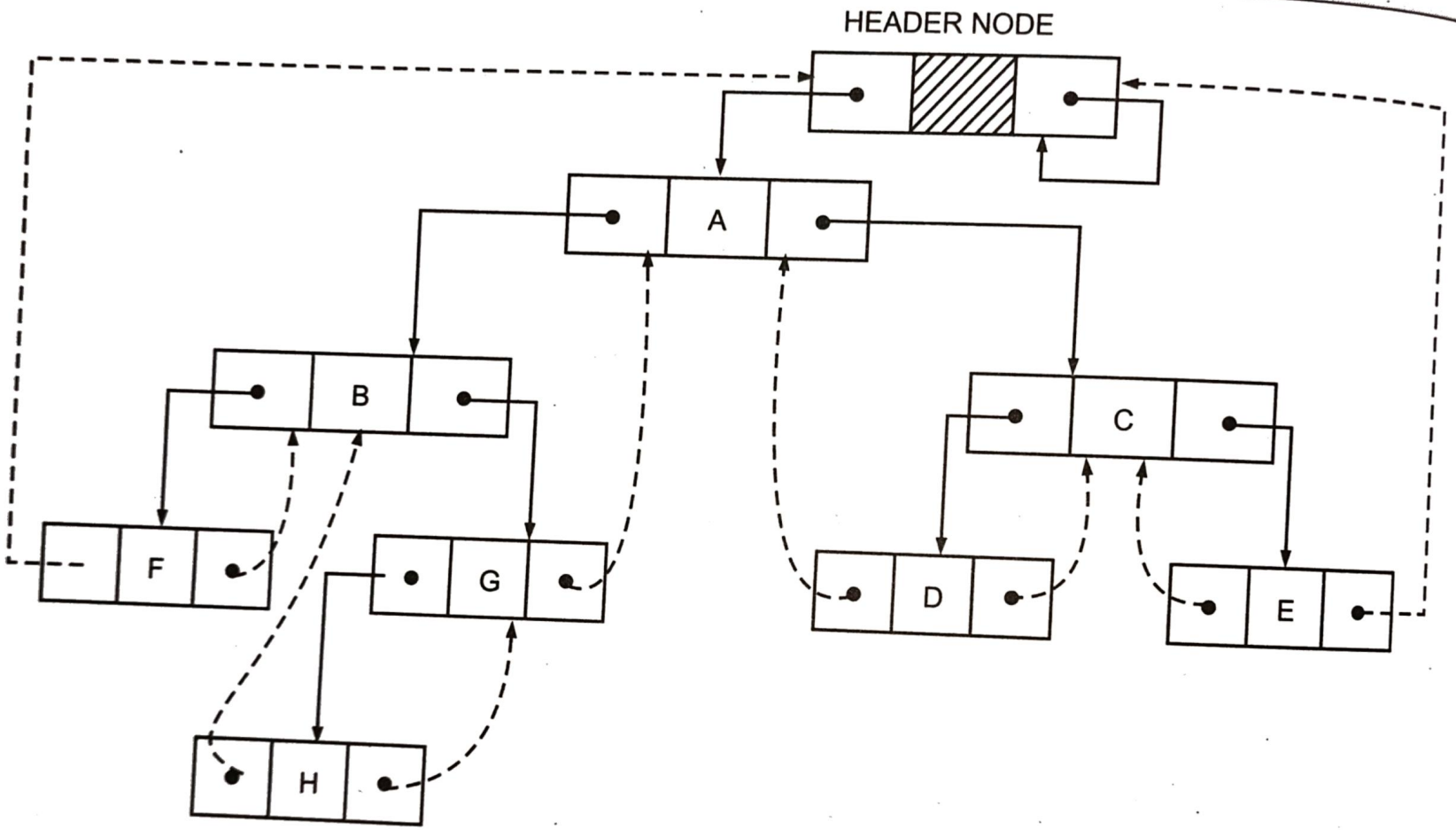


Fig. 8.66.

8.16 TRAVERSING IN

```
                ptr=ptr->rightptr;
            }
        }
    }
```

8.17 INSERTION AND DELETION IN A THREADED BINARY TREE

(a) Insertion in a threaded binary search tree

As in binary search tree here also we will find the exact location where we have to insert the new node.

The new node will be inserted as a leaf node so its left and right pointers both will be threads.

```
tmp->leftptr=thread;
tmp->rightptr=thread;
```

Case 1: When the tree is empty.

When the tree is empty the left pointer of the head node is a thread pointing to itself. We will insert the new node as the left child of header node so now the left pointer of the head will be a link pointing to the new node. We know that the left pointer of first node in inorder traversal and the right pointer of last node in inorder traversals are threads pointing to the header node. Here we have only one node which is the first and the last node so its left right pointers will be threads pointing to the header node.

```
head->left=link;
head->leftptr=tmp;
tmp->leftptr=head;
tmp->rightptr=head;
```

Case 2: When new node inserted as the left child of its parent.

The threads of new node will point to its inorder prodecessor and successor. The node which was inorder predecessor of the parent is now the inorder predecessor of this node. The inorder successor of this node is its parent.

```
tmp->lchild=parent->lchild;
tmp->rchild=parent;
```

The parent of new node has a thread in its left pointer pointing to its predecessor, but after insertion its left pointer will be a link pointing to the new node.

```
parent->left=link;
parent->lchild=tmp;
```

Case 3: When new node is inserted as the right child of its parent.

The node which was inorder sucessor of the parent is now the inorder sucessor of this node. The inorder predecessor of this node is its parent. So the left and right threads of the new node will be:

```
tmp->leftptr=parent;
tmp->rightptr=parent->child;
```

The parent of new node had a thread in its right pointer pointing to its successor, but after insertion its right pointer will be a link pointing to the new node.

```
parent->right=link;
parent->rchild=tmp;
```

(b) Deletion from a threaded binary search tree

As in binary search tree here also we have five possibilites while performing delet operation

1. There are no nodes in the tree *i.e.*, the tree is empty.
2. Node to be deleted is not present in the tree.
3. Node to be deleted is leaf node *i.e.*, it has no children.
4. Node to be deleted has only one child.
5. Node to be deleted has two children.

The last three possibilities require the manipulation of pointers so we will discuss them.

Case A: Node to be deleted is leaf node.

If the node to be deleted is the root node of the tree then the tree will become empty after its deletion so then left pointer of head will be a thread pointing to itself.

```
head->left=thread;
head->leftptr=head;
```

If the node to be deleted is a left leaf node then the left pointer of parent will become a thread pointing to its inorder predecessor. Initially its inorder predcessor was its left child but now its inorder predecessor will be that node which was predecessor of its left child.

```
par->left=thread;
par->leftptr=loc->leftptr;
```

If the node to be deleted is a right leaf node then the right pointer of parent will become a thread pointing to its inorder successor. Initially its inorder successor was its right child but now its inorder successor will be that node which was successor of its right child.

```
par->right=thread;
par->rightptr=loc->rightptr;
```

Case B: Node to be deleted has one child.

Delete the node as in binary search tree. Find the inorder successor and inorder predecessor of the node to be deleted.

```
S = inorder_successor(loc);
p = inorder_precessor(loc);
```

If node has right subtree then put the predecessor of the node in the left pointer of its successor.

```
if(loc->right==link)
        s->leftptr=p;
```

If node has left subtree then put the successor of the node in the right pointer of its predecessor.

```
If(loc->left==link)
    p->rightptr=s;
```

Case C: Node to be deleted has two children.

Here first we delete the inorder successor of node and then replace the deleted node with the inorder successor. For deleting inorder successor we will call either case *A* or case *B*.