**Unit Rationale**

|  |  |  |
| --- | --- | --- |
| **Unit I :** | **TREES** | **09 Hours** |
| Tree- basic terminology, General tree and its representation, representation using sequential and linked organization, Binary tree- properties, converting tree to binary tree, binary tree traversals- inorder, preorder, post order, level wise -depth first and breadth first, Operations on binary tree. Binary Search Tree (BST), BST operations, Threaded binary tree- concepts, threading, insertion and deletion of nodes in in-order threaded binary tree, in order traversal of in-order threaded binary tree. Case Study- Use of binary tree in expression tree-evaluation and Huffman's coding | | |
|  | | |

**Unit II : Graphs 09Hours**

Basic Concepts, Storage representation, Adjacency matrix, adjacency list, adjacency multi list,inverse adjacency list. Traversals-depth first and breadth first, Introduction to Greedy Strategy, Minimum panning Tree, Greedy algorithms for computing minimum spanning tree-Prims and Kruskal algorithm -s, Dikjtra's Single source shortest path, Topological ordering.Case study-Data structure used in Webgraph and Google map.

**Unit III: Hashing 09Hours**

Hash Table-Concepts-ash table, hash function, bucket, collision, probe, synonym, overflow, open hashing, closed hashing, perfect hash function, load density, full table, load factor, rehashing, issues in hashing, hash functions-properties of good hash function, division, multiplication, extraction, mid-square, folding and universal, Collision resolution strategies-open addressing and chaining, Hash table overflow-open addressing and chaining, extendible hashing. Dictionary-Dictionary as ADT, ordered dictionaries. Skip List-representation, searching andoperations-insertion, removal.

**Unit IV: Search Trees 09Hours**

Symbol Table-Representation of Symbol Tables-Static tree table and Dynamic tree table,Introduction to Dynamic Programming, Weight balanced tree, Optimal Binary Search Tree (OBST), OBST as an example of Dynamic Programming, Height Balanced Tree-AVL tree.

**Unit V: Indexing and Multiway Trees 09Hours**

Indexing and Multiway Trees-Indexing, indexing techniques, Types of search tree-Multiway search tree, B-Tree, B+Tree, Trie Tree, Splay Tree, Red-Black Tree, K-dimensional tree, AA tree. Set-Set ADT, realization of Set and operations.Heap-Basic concepts, realization of heap and operations, Heap as a priority queue, heap sort.

**Unit VI: File Organization 09Hours**

Sequential file organization-concept and primitive operations, Direct Access File-Concepts and

Primitive operations, Indexed sequential file organization-concept, types of indices, structure of

index sequential file, Linked Organization-multi list files, coral rings, inverted files and cellular

partitions.External Sort-Consequential processing and merging two lists, multiday merging- a k way merge algorithm.

**UNIT I**

**TREES**

**Unit Rationale**

**Unit 1: (9 Hrs)**

Tree-basic terminology, General tree and its representation, representation using sequential and linked organization, Binary tree -properties, converting tree to binary tree, binary tree traversals-inorder, preorder, post order, level wise -depth first and breadth first, Operations on binary tree. Binary Search Tree (BST), BST operations, threaded binary tree-concepts, threading, insertion and deletion of nodes in in-order threaded binary tree, in order traversal of in-order threaded binary tree. Case Study-Use of binary tree in expression tree-evaluation and Huffman's coding

**Upon completion Students will be able to: -**

When the students have successfully completed this course, they will be able to:

**Tlo need to add**

|  |  |  |  |
| --- | --- | --- | --- |
| **Session** | **Contents** | **Objective** | **Outcome** |
| 1 | **Tree-** basic terminology, General tree | Understand tree data structure to store the data. | Apply tree data structure concept in real life applications |
| 2 | Representation using sequential and linked organization, Binary tree- properties | Understand the properties of Binary tree. | Identify problems and use appropriate technique |
| 3 | converting tree to binary tree, **binary tree traversals-** inorder, preorder, post order,  level wise -depth first and breadth first, | Concerting a tree to binary tree and finding tree traversals | Identify and use operators according to their placement in the hierarchy chart. |
| 4 | Operations on binary tree. Binary Search Tree (BST) | Use algorithms, flowcharts, and pseudo code to develop the instructions for  Each module in the solution of a problem. |  |
| **5** | BST operations | Use flow chart for testing & coding the program | Apply flow chart for testing & coding |
| **6** | Threaded binary tree- concepts, threading, insertion of nodes in in-order threaded binary tree | Evaluate time complexity & space | Identify the time & space complexity of algorithm |
| **7** | Threaded binary tree- deletion of nodes in in-order threaded binary tree | Use the sorting method concepts for solving problem on computer | Identify the sorting method for solving the problem on computer |
| **8** | in order traversal of in-order threaded binary tree. | Use different method for sorting | Identify the sorting method |
| **9** | **Case Study**- Use of binary tree in expression tree-evaluation and Huffman's coding |  |  |

**SESSION 1**

**Session 1: Tree- basic terminology, General tree.**

**Session Plan**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

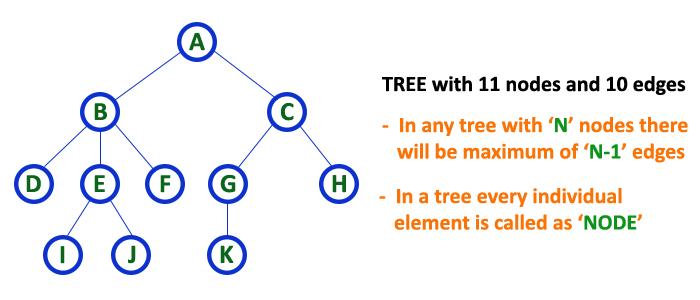
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time (in min)** | **Content** | **Learning Aid/Methodology** | **Faculty Approach** | **Typical Student Activity** | **Skill /Competency Developed** |
| 10 | Introduction | Presentation and Quiz | Introduces  Facilitates  monitors | Listens  Participate  Discuss | Knowledge |
| 15 | basic terminology | Brain storming | Facilitates | Listens  Participate  Discuss | Comprehension  Application |
| 05 | General tree | Brain storming | Explain | Listen | Knowledge |
| 25 | General tree representation | Brain storming  Quiz | Explain  Facilitates | Listens | Comprehension  Application |
| 5 | Conclusion and Summary | Key Words | List  Facilitates | Identifies | Knowledge  Comprehension |

**NOTES ­­­­­­­­­­­**

1.1 Tree- basic terminology,

Root: Root is a specially designed node (or data items) in a tree. It is the first node in the hierarchical arrangement of the data items. ‘A’ is a root node in the Fig. 1 Each data item in a tree is called a node . It specifies the data information and links (branches) to other data items

node:In tree data structure, 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.  
  
In a tree data structure, if we have N number of nodes then we can have a maximum of N-1 number of links.



# 1. Root

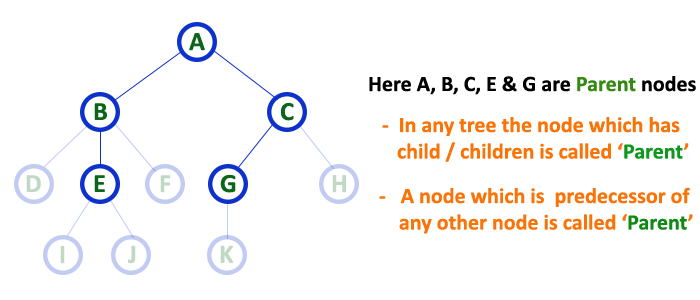
In a tree data structure, the first node is called as Root Node. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

# 2. Edge

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

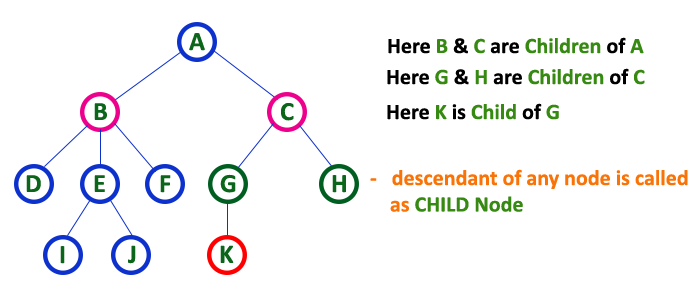
# 3. Parent

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



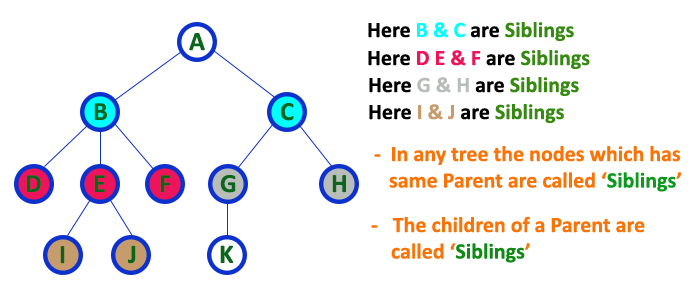
# 4. Child

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.



# 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.

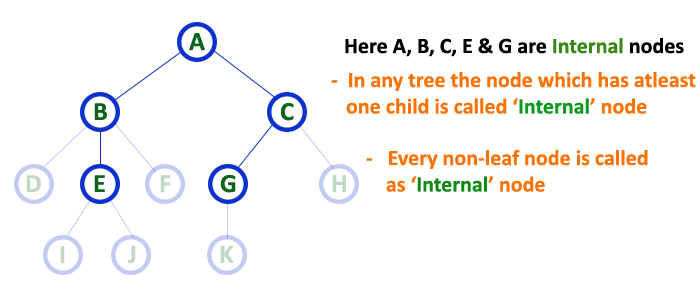


# 6. Leaf

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.

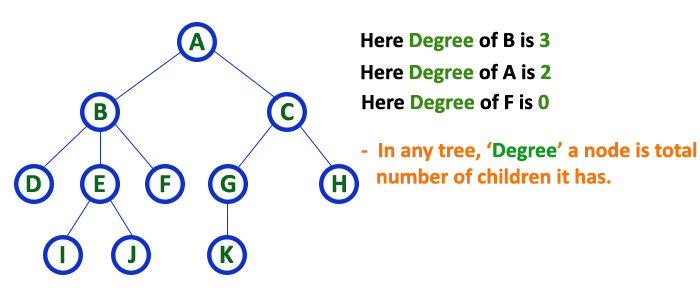
# 7. Internal Nodes

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.



# 8. Degree

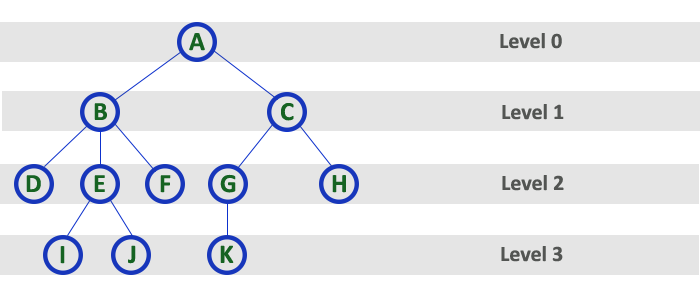
In a tree data structure, the total number of children of a node is called as DEGREE of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'



# The number of branches associated with a node is called the "degree of the node". -> When the branch is directed toward a node, it is an "indegree branch" ; when the branch is directed away from the node, it is called "outdegree branch". -> The sum of indegree branches and outdegree branches is called the "degree of the node. -> The indegree of the root is zero by definition.

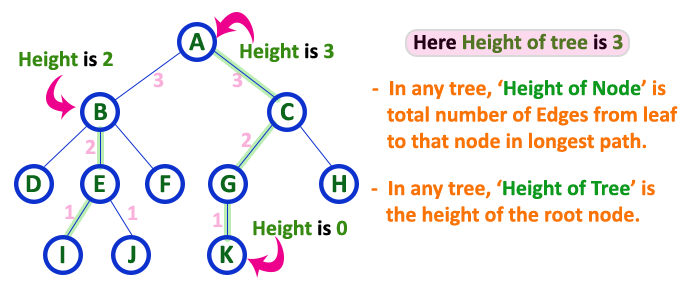
# 9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In 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'.



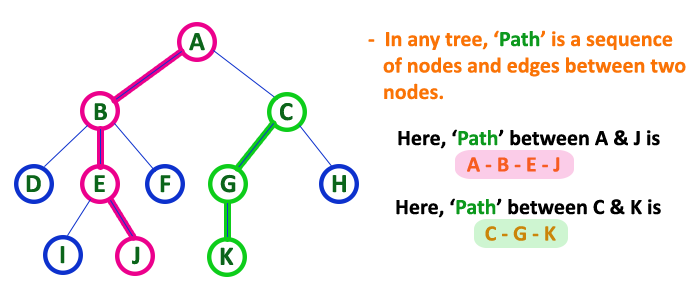
# 11. Depth

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'.

# 

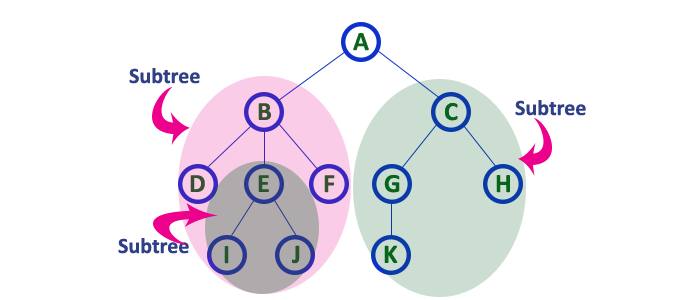
# 12. Path

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



# 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.

14. Descendant

A node reachable by repeated proceeding from parent to child.

15. Ancestor

A node reachable by repeated proceeding from child to parent.

16. Level

The level of a node is defined by 1 + (the number of connections between the node and the root).

17. Height of node

The height of a node is the number of edges on the longest path between that node and a leaf.

18. Height of tree

The height of a tree is the height of its root node.

19. Depth

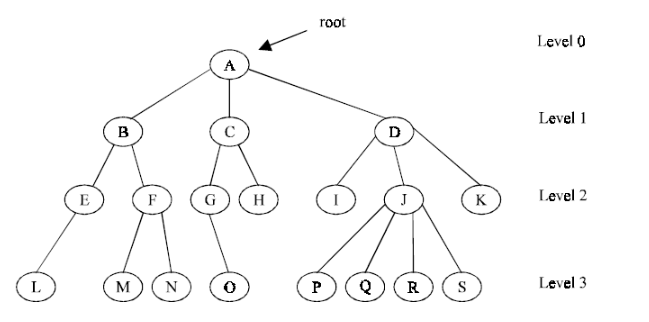
The depth of a node is the number of edges from the tree's root node to the node.

20. Forest

A forest is a set of n ≥ 0 disjoint trees.

**1.2 General tree and its representation**

Trees are very flexible, versatile and powerful non - liner data structure that can be used to represent data items possessing hierarchical relationship between the grand father and his children and grand children as so on.

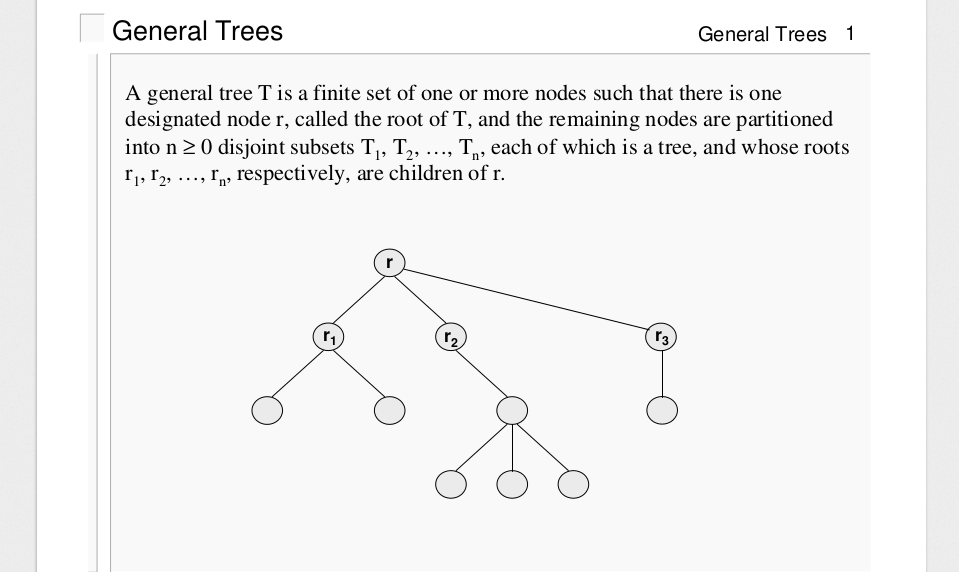
Fig.1:general tree

A tree is an ideal data structure for representing hierarchical data. A tree can be theoretically defined as a finite set of one or more data items (or nodes) such that :

1. There is a special node called the root of the tree.

2. Removing nodes (or data item) are partitioned int o number of mutually exclusive ( i.e., disjoined) subsets each of which is itself a tree, are called sub tree.

A Tree in which each node having either 0 or more child nodes is called general tree. So we can say that a Binary Tree is a specialized case of General tree.   
General Tree is used to implement File System.



**SESSION 2**

**Session 2:** Representation using sequential and linked organization, Binary tree- properties

**Session Plan**

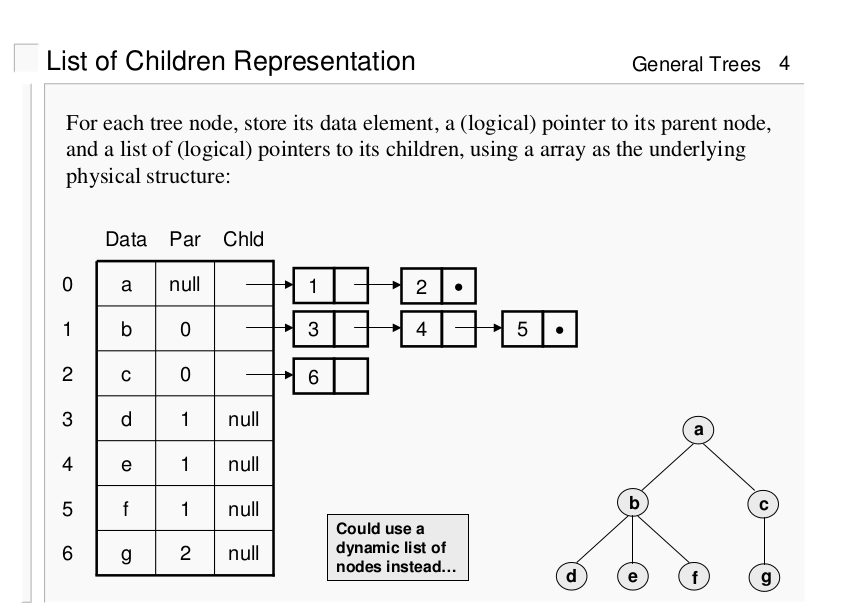
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

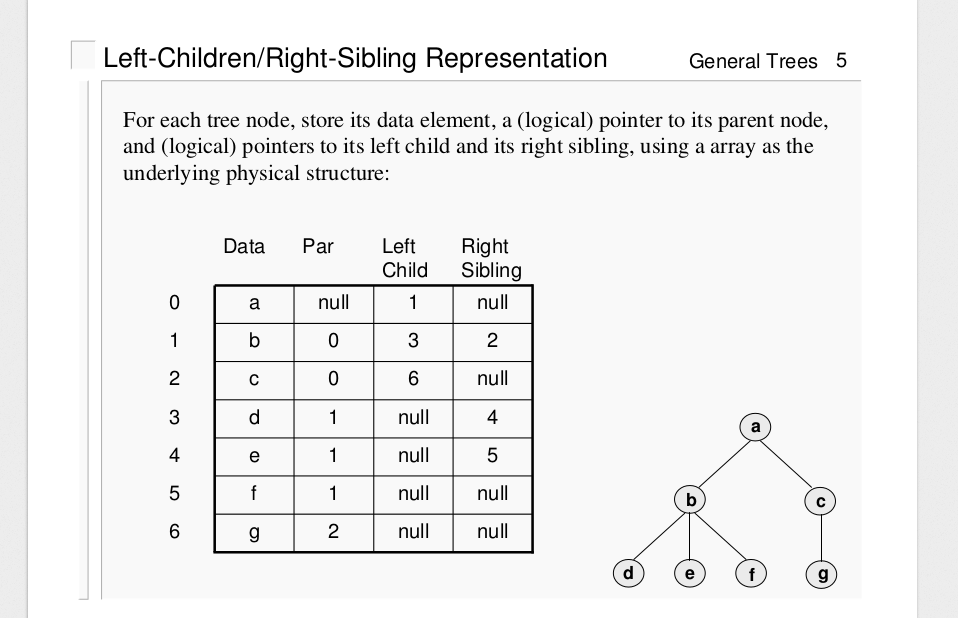
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time (in min)** | **Content** | **Learning Aid/Methodology** | **Faculty Approach** | **Typical Student Activity** | **Skill /Competency Developed** |
| 05 | Introduction & revision | Presentation and Quiz | Introduces  Facilitates  monitors | Listens  Participate  Discuss | Knowledge |
| 15 | Representation using sequential organization | Brain storming | Facilitates | Listens  Participate  Discuss | Comprehension  Application |
| 15 | linked organization | Brain storming | Explain | Listen | Knowledge |
| 20 | Binary tree- properties | Brain storming  Quiz | Explain  Facilitates | Listens | Comprehension  Application |
| 5 | Conclusion and Summary | Key Words | List  Facilitates | Identifies | Knowledge  Comprehension |

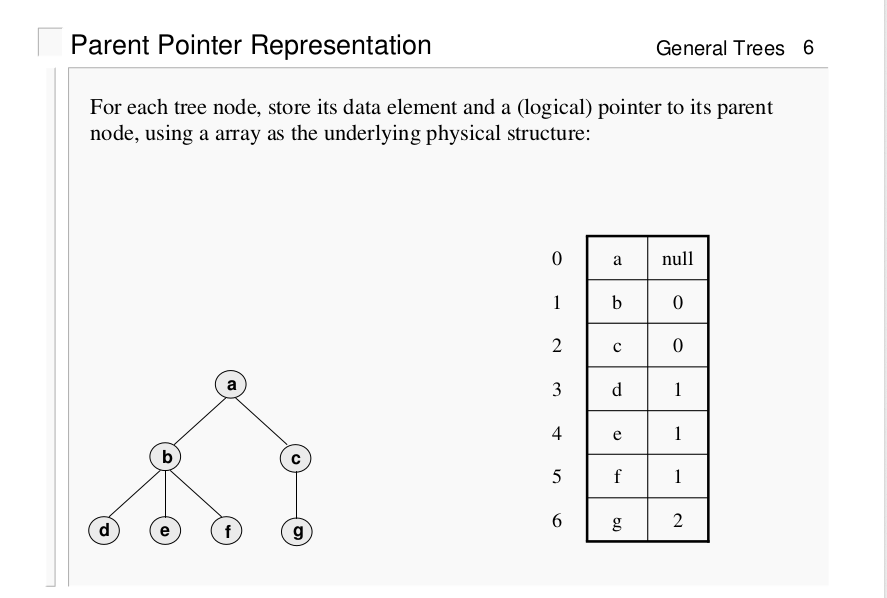
**NOTES**

**----------------------------------------------------------------------------------------**

**Representing General Trees**

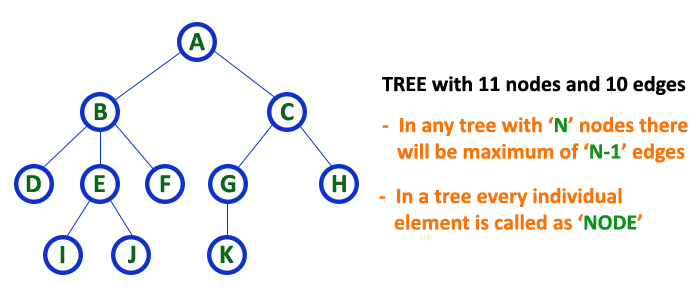






**Example**

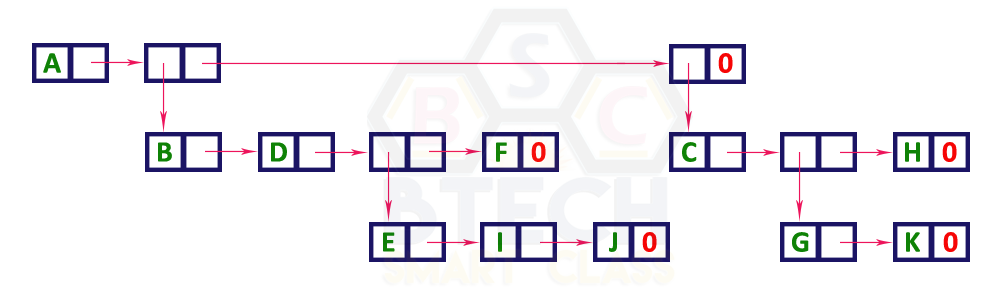
**Consider following tree**



# 1. List Representation

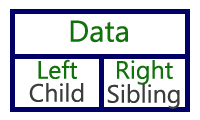
In this representation, we use two types of nodes one for representing the node with data and another for representing only references. We start with a node with data from root node in the tree. Then it is linked to an internal node through a reference node and is linked to any other node directly. This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

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

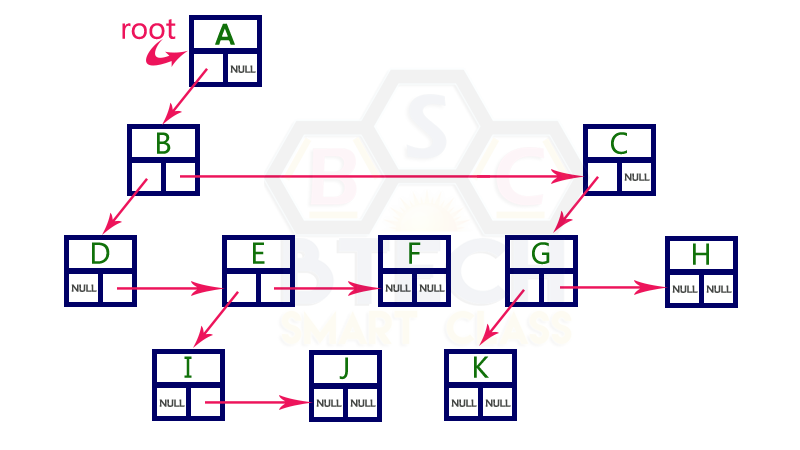


# 2. Left Child - Right Sibling Representation

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. Data field stores the actual value of a node, left reference field stores the address of the left child and right reference field stores the address of the right sibling node. Graphical representation of that node is as follows...



in this representation, every node's data field stores the actual value of that node. If that node has left child, then left reference field stores the address of that left child node otherwise that field stores NULL. If that node has right sibling then right reference field stores the address of right sibling node otherwise that field stores NULL.   
  
The above tree example can be represented using Left Child - Right Sibling representation as follows...



**Difference between general tree and binary tree**

|  |  |
| --- | --- |
| General tree | Binary tree |
| A general tree is a data structure in that each node can have n number of children, | A Binary tree is a data structure in that each node has at most two nodes left and right. |
|  |  |
| Root  In-degree 0  Out-degree n(max) | Root  In-degree 0  Out-degree 2(max) |
| Node  In-degree 1  Out-degree n(max) | Node  In-degree 1  Out-degree 2(max) |
| Leaf Node  In-degree 1  Out-degree 0 | Leaf Node  In-degree 1  Out-degree 0 |
| There is no limit on the degree of node in a general tree. | Nodes in a binary tree cannot have more than degree 2. |
| Height of a general tree is the length of longest path from root to the leaf of tree. Height(T) = {max(height(child1) , height(child2) , … height(child-n) ) +1} | Height of a binary tree is : Height(T) = { max (Height(Left Child) , Height(Right Child) + 1} |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Properties of Binary Tree**

* + **The maximum number of nodes on level *i* of a binary tree is 2*i* -1, *i* ≥ 1.**
  + **The maximum number of nodes in a binary tree of depth *k* is is 2*k* -1, *k* ≥ 1.**
  + **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.**
  + **A *full binary tree* of depth *k* is a binary tree of depth *k* having 2*k* -1 nodes, *k* ≧ 0.**
  + **A binary tree with *n* nodes and depth *k* is *complete* *iff* its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of depth *k*.**

**SESSION 3**

**Session 3:** converting tree to binary tree, **binary tree traversals-** inorder, preorder, post order,

level wise -depth first and breadth first,

**Session Plan**

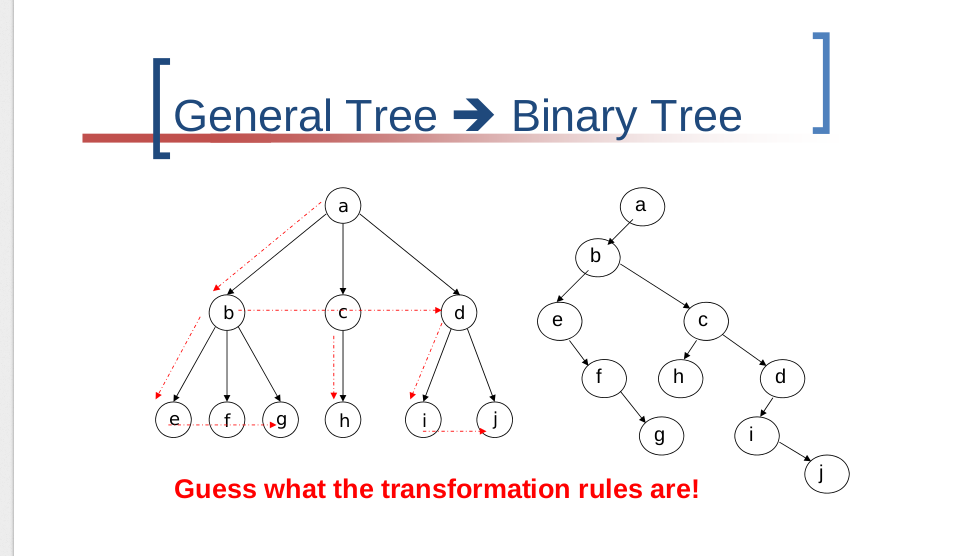
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time (in min)** | **Content** | **Learning Aid/Methodology** | **Faculty Approach** | **Typical Student Activity** | **Skill /Competency Developed** |
| 05 | Introduction & revision | Presentation and Quiz | Introduces  Facilitates  monitors | Listens  Participate  Discuss | Knowledge |
| 15 | converting tree to binary tree | Brain storming | Facilitates | Listens  Participate  Discuss | Comprehension  Application |
| 15 | **binary tree traversals-** inorder, preorder, post order, | Brain storming | Explain | Listen | Knowledge |
| 20 | level wise -depth first and breadth first, | Brain storming  Quiz | Explain  Facilitates | Listens | Comprehension  Application |
| 5 | Conclusion and Summary | Key Words | List  Facilitates | Identifies | Knowledge  Comprehension |

**NOTES**

**----------------------------------------------------------------------------------------**

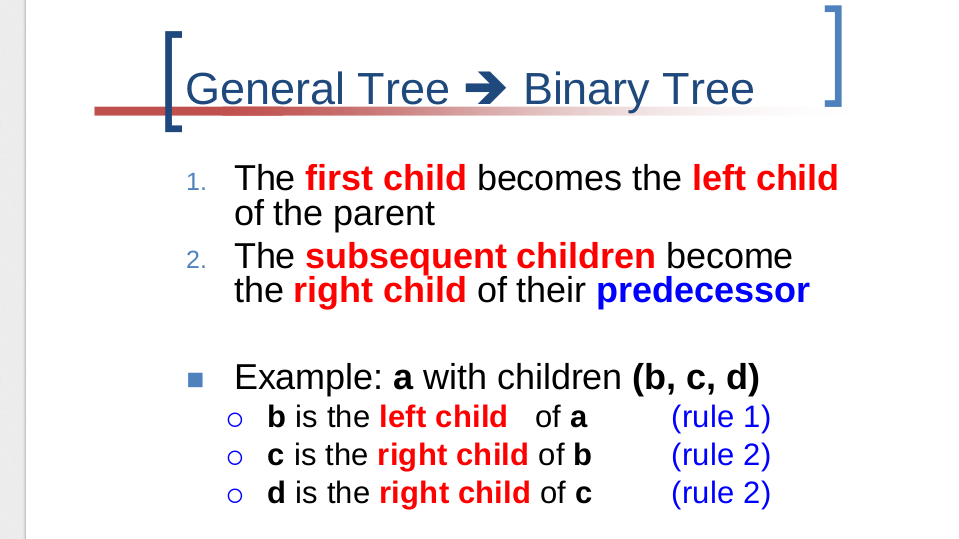
**General tree to binary tree conversion.**

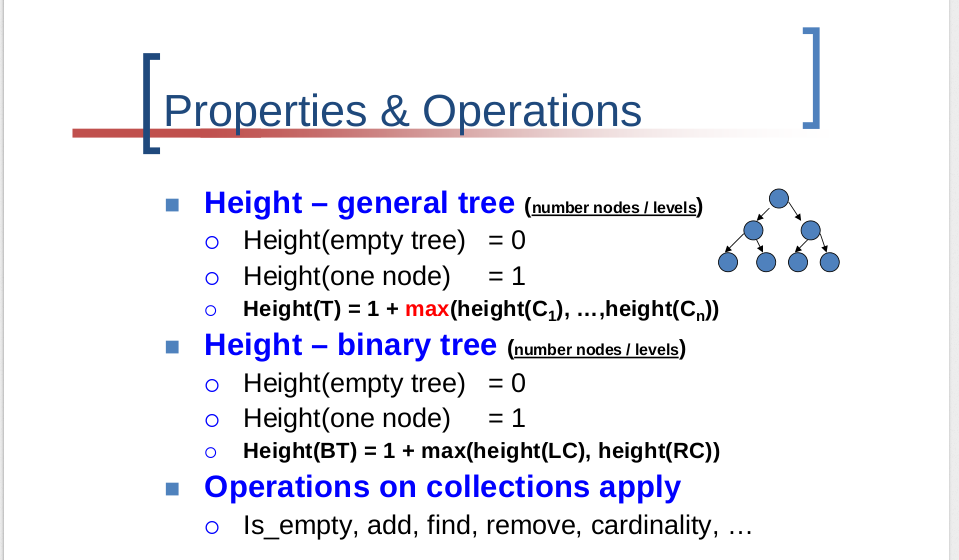


Steps:

The process of converting general tree in to binary tree is given below:

1. Root node of general tree becomes root node of Binary Tree.
2. Now consider T1, T2, T3 ... Tn are child nodes of the root node in general tree. The left most child (T1) of the root node in general tree becomes left most child of root node in the binary tree. Now Node T2 becomes right child of Node T1, Node T3 becomes right child of Node T2 and so on in binary tree.
3. The same procedure of step 2 is repeated for each leftmost node in the general tree.





**Binary tree-**

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called as Binary Tree.

n a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

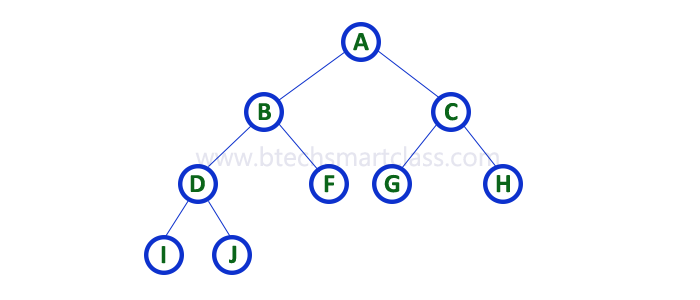
# Example

# 

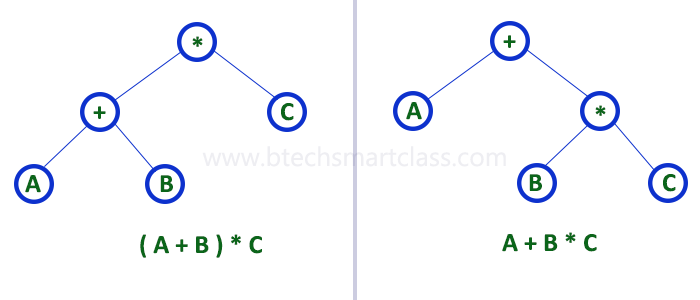
**There are different types of binary trees and they are...**

# 1. Strictly Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows...A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree. Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree



Strictly binary tree data structure is used to represent mathematical expressions.

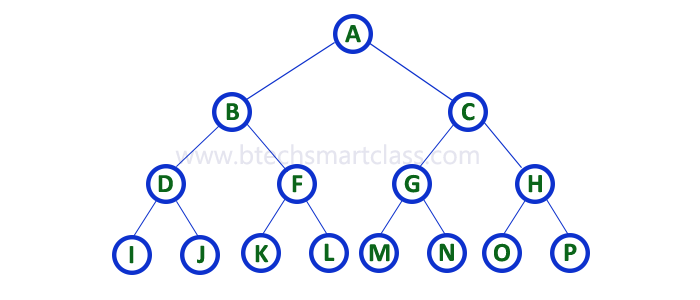


# 2. Complete Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2level number of nodes. For example at level 2 there must be 22 = 4 nodes and at level 3 there must be 23 = 8 nodes.

A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.

Complete binary tree is also called as Perfect Binary Tree

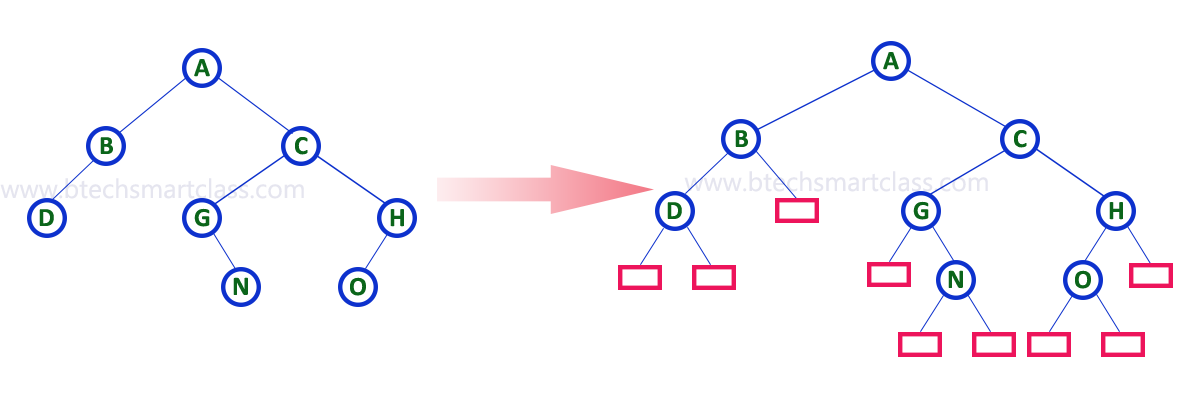


# 3. Extended Binary Tree

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.

The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.

In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes

****

# Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

1. Array Representation
2. Linked List Representation

Consider the following binary tree...



# 1. Array Representation

In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree.  
Consider the above example of binary tree and it is represented as follows...



To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of 2n+1 - 1.

# 2. Linked List Representation

We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.  
In this linked list representation, a node has the following structure...



he above example of binary tree represented using Linked list representation is shown as follows...



**binary tree traversals-inorder, preorder, post order:**

When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method.

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

1. In - Order Traversal
2. Pre - Order Traversal
3. Post - Order Traversal

Consider the following binary tree...



# 1. In - Order Traversal ( leftChild - root - rightChild )

In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.  
  
In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the left most child. So first we visit 'I' then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit 'G' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node 'C' and next visit C's right child 'H' which is the right most child in the tree so we stop the process.  
  
That means here we have visited in the order of I - D - J - B - F - A - G - K - C - H using In-Order Traversal.

In-Order Traversal for above example of binary tree is I - D - J - B - F - A - G - K - C - H

# 2. Pre - Order Traversal ( root - leftChild - rightChild )

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.   
  
In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the left most child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this we have completed node C's root and left parts. Next visit C's right child 'H' which is the right most child in the tree. So we stop the process.  
  
That means here we have visited in the order of A-B-D-I-J-F-C-G-K-H using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is

#### A - B - D - I - J - F - C - G - K – H

Algorithm preOrder(v)

visit(v)

for each child w of v

preorder(w)

# 2. Post - Order Traversal ( leftChild - rightChild - root )

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.  
  
Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

#### I - J - D - F - B - K - G - H - C – A

AlgorithmpostOrder(v)

for each child w of v

postOrder(w)

visit(v)

**level wise -depth first and breadth first,**

A Tree is typically traversed in two ways:

* [Breadth First Traversal (Or Level Order Traversal)](http://www.geeksforgeeks.org/level-order-tree-traversal/)
  + Visit each node level wise.
* [Depth First Traversals](http://www.geeksforgeeks.org/618/)
  + Inorder Traversal (Left-Root-Right)
  + Preorder Traversal (Root-Left-Right)
  + Postorder Traversal (Left-Right-Root)



BFS and DFSs of above Tree

Breadth First Traversal : 1 2 3 4 5

Depth First Traversals:

Preorder Traversal : 1 2 4 5 3

Inorder Traversal : 4 2 5 1 3

Postorder Traversal : 4 5 2 3 1

**Why do we care?**  
There are many tree questions that can be solved using any of the above four traversals. Examples of such questions are [size](http://www.geeksforgeeks.org/write-a-c-program-to-calculate-size-of-a-tree/), [maximum](http://geeksquiz.com/find-maximum-or-minimum-in-binary-tree/), [minimum](http://geeksquiz.com/find-maximum-or-minimum-in-binary-tree/), [print left view](http://www.geeksforgeeks.org/print-left-view-binary-tree/), etc.

**Is there any difference in terms of Time Complexity?**  
All four traversals require O(n) time as they visit every node exactly once.

**SESSION 4**

**Session 4:** Operations on Binary Search Tree (BST)

**Session Plan**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time (in min)** | **Content** | **Learning Aid/Methodology** | **Faculty Approach** | **Typical Student Activity** | **Skill /Competency Developed** |
| 05 | Introduction & revision | Presentation and Quiz | Introduces  Facilitates  monitors | Listens  Participate  Discuss | Knowledge |
| 10 | Binary Search Tree (BST) | Brain storming | Facilitates | Listens  Participate  Discuss | Comprehension  Application |
| 20 | Operations on binary tree: insert | Brain storming | Explain | Listen | Knowledge |
| 20 | Operations on binary tree: insert | Brain storming  Quiz | Explain  Facilitates | Listens | Comprehension  Application |
| 5 | Conclusion and Summary | Key Words | List  Facilitates | Identifies | Knowledge  Comprehension |

**NOTES:**

**Binary Search Tree (BST)**

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

* The left sub-tree of a node has a key less than or equal to its parent node's key.
* The right sub-tree of a node has a key greater than or equal to its parent node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

left\_subtree (keys) ≤ node (key) ≤ right\_subtree (keys)

## Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST −

We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

## Basic Operations

Following are the basic operations of a tree −

* **Search** − Searches an element in a tree.
* **Insert** − Inserts an element in a tree.
* **Pre-order Traversal** − Traverses a tree in a pre-order manner.
* **In-order Traversal** − Traverses a tree in an in-order manner.
* **Post-order Traversal** − Traverses a tree in a post-order manner.

## Node

Define a node having some data, references to its left and right child nodes.

struct node {

int data;

struct node \*leftChild;

struct node \*rightChild;

};

## Search Operation

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

### Algorithm

struct node\* search(int data){

struct node \*current = root;

printf("Visiting elements: ");

while(current->data != data){

if(current != NULL) {

printf("%d ",current->data);

//go to left tree

if(current->data > data){

current = current->leftChild;

}//else go to right tree

else {

current = current->rightChild;

}

//not found

if(current == NULL){

return NULL;

}

}

}

return current;

}

## Insert Operation

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

### Algorithm

void insert(int data) {

struct node \*tempNode = (struct node\*) malloc(sizeof(struct node));

struct node \*current;

struct node \*parent;

tempNode->data = data;

tempNode->leftChild = NULL;

tempNode->rightChild = NULL;

//if tree is empty

if(root == NULL) {

root = tempNode;

} else {

current = root;

parent = NULL;

while(1) {

parent = current;

//go to left of the tree

if(data < parent->data) {

current = current->leftChild;

//insert to the left

if(current == NULL) {

parent->leftChild = tempNode;

return;

}

}//go to right of the tree

else {

current = current->rightChild;

//insert to the right

if(current == NULL) {

parent->rightChild = tempNode;

return;

}

}

}

}

}

**SESSION 5**

**Session 5:** BST operations

**Session Plan**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time (in min)** | **Content** | **Learning Aid/Methodology** | **Faculty Approach** | **Typical Student Activity** | **Skill /Competency Developed** |
| 05 | Introduction & revision | Presentation and Quiz | Introduces  Facilitates  monitors | Listens  Participate  Discuss | Knowledge |
| 10 | Binary Search Tree (BST) | Brain storming | Facilitates | Listens  Participate  Discuss | Comprehension  Application |
| 20 | Operations on binary tree: delete | Brain storming | Explain | Listen | Knowledge |
| 20 | Operations on binary tree: delete | Brain storming  Quiz | Explain  Facilitates | Listens | Comprehension  Application |
| 5 | Conclusion and Summary | Key Words | List  Facilitates | Identifies | Knowledge  Comprehension |

**NOTES:**

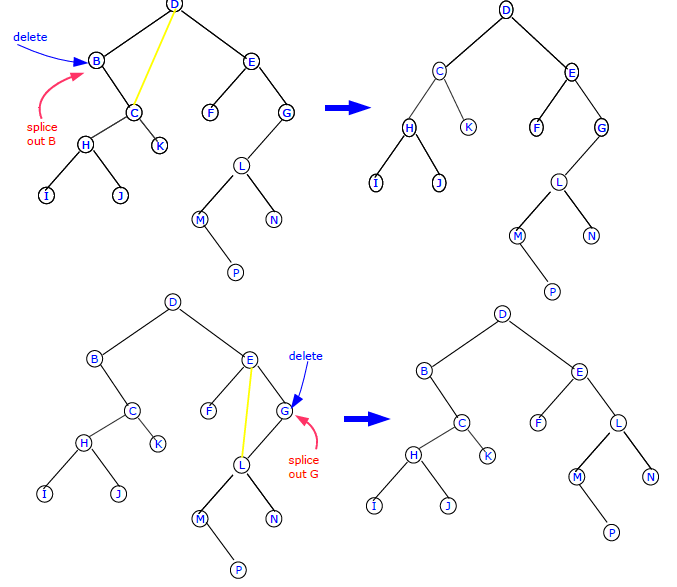
**Binary Search Tree Deletion operation**

1. To delete a leaf node, just delete it.

2. If the node to be deleted has only one child, splice that node out by connecting its parent and

child as shown below:

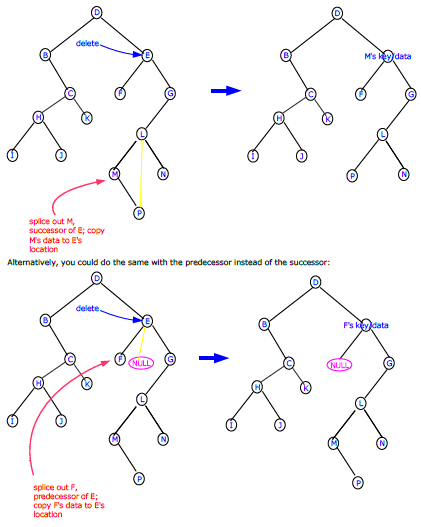
(Note: in the figures below, the letters in the nodes are just labels, not KEY values)



3. If the node to be deleted has two children, splice out its successor, and replace the key/data in

the node to be deleted by the key/data from the spliced out successor:

Alternatively, you could do the same with the predecessor instead of the successor:



/\*\*\*\*\*\*\*\*\*Deletion Algorithm - From Carmen/Leiserson/Rivest/Stein\*\*\*\*\*\*\*\*\*\*\*\*

delete(T, z):

//First determine node y to splice out

if left[z]==NULL or right[z]==NULL

y=z //node to splice out if z has AT MOST one child

else

y=successor(z) //node to splice out if z has two children

//set the non-NULL child of y to x, or set x to NULL if y has no children

if left[y] != NULL

x=left[y]

else

x=right[y]

//Now do the splicing

if x != NULL

parent[x]=parent[y]

if parent[y]==NULL

x=root[T]

else

if y==left[parent[y]]

x=left[parent[y]]

else

x=right[parent[y]]

//Splicing done

// if successor of z was the spliced out node, move y's key/data to z

if y != z

key[y]=key[z]]

//copy non-key data too if present.

//return y if needed

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Here is a complete Binary Search Tree program that implements the deletion operation.

/\*BTree.cpp \*/

#include<iostream>

using namespace std;

class BTree

{

public:

int key; //data in the node

BTree \*left; // Pointer to the left subtree.

BTree \*right; // Pointer to the right subtree.

BTree \*parent; // Pointer to the parent node

BTree();

~BTree();

void insert(int key); //insert a new node at a leaf position with the given int data

void insert(BTree\* leaf); //insert a given leaf node into the tree

BTree \*search(int key); //return NULL if no node has given int value, or a pointer

to the node that has the int value

void destroy(); //clean up the whole tree

const void preOrderPrint();

const void inOrderPrint();

const void postOrderPrint();

BTree\* find\_Max();

BTree\* find\_Min();

int countNodes();

int countLeafNodes();

int find\_depth();

BTree\* successor(int); //Returns pointer to successor node of this node

BTree\* predecessor(int) ; //Returns pointer to predecessor node of this node

void remove(int val); //delete the node pointed to by the pointer Node.

BTree\* find\_root(); //returns a pointer to the root node

};

BTree::BTree(){

left=NULL;

right=NULL;

parent=NULL;

}

void BTree::insert(int key){

BTree\* bt=new BTree; //new node

bt->key=key; //assign key to new node

insert(bt);

}

void BTree::insert( BTree\* leaf){

if(!(this->key)){

key=leaf->key;

}

else if ( (this->key) >= (leaf->key) ){

if((this->left != NULL)){

(this->left)->insert(leaf);

}else{

this->left=leaf;

leaf->parent=this;

}

}

else {

if(this->right != NULL){

(this->right)->insert(leaf);

}else{

this->right=leaf;

leaf->parent=this;

}

}

}

const void BTree::inOrderPrint(){

if( (left==NULL) && (right==NULL) ){

if(key){cout<<key<<" ";}

}

else {

if(left)left->inOrderPrint();

cout<<key<<" ";

if(right)right->inOrderPrint();

}

}

const void BTree::preOrderPrint(){

if( (left==NULL) && (right==NULL) ){

if(key){cout<<key<<" ";}

}

else{

cout<<key<<" ";

if(left)left->preOrderPrint();

if(right)right->preOrderPrint();

}

}

const void BTree::postOrderPrint(){

if( (left==NULL) && (right==NULL) ){

if(key){cout<<key<<" ";}

}

else{

if(left)left->postOrderPrint();

if(right)right->postOrderPrint();

cout<<key<<" ";

}

}

BTree\* BTree::search(int keyval){

if(this->key==keyval){

return this;

}

else if( (this->key) > keyval){ //search in left subtree

if(this->left)

return this->left->search(keyval);

else return NULL;

}else{

if(this->right)

return this->right->search(keyval);

else return NULL;

}

}

BTree\* BTree::find\_Max(){

if(right == NULL){

return this;

}else{

right->find\_Max();

}

}

BTree\* BTree::find\_Min(){

if(left == NULL){

return this;

}else{

left->find\_Min();

}

}

int BTree::countNodes(){

int l, r;

if((left==NULL)&&(right==NULL)){

if(key)return 1; else return 0;

}

else{

if(left)

l=left->countNodes(); else l=0;

if(right)

r=right->countNodes(); else r=0;

return 1+l+r;

}

}

BTree\* BTree::find\_root(){ //return root of the tree

if (this->parent == NULL)

return this;

else return (this->parent)->find\_root();

}

int BTree::countLeafNodes(){

int c=0,l=0,r=0;

if((left==NULL)&&(right==NULL)){

if(key) c++;

}else{

if(left)

l=left->countLeafNodes();

if(right)

r=right->countLeafNodes();

}

return c+l+r;

}

int BTree::find\_depth(){

if(this==NULL)

return 0;

else{

int leftDepth=left->find\_depth();

int rightDepth=right->find\_depth();

if (leftDepth>=rightDepth)

return 1+leftDepth;

else

return 1+rightDepth;

}

}

/\*\*\*\*\*\*\*\*\*\*\*Algorithm-Cormen/Leiserson/Rivest/Stein\*\*\*\*\*\*\*\*\*

TREE-SUCCESSOR(x)

if(right[x] != NULL)

return find\_min(right[x]);

y=parent[x];

while ( y != NULL and x == right[y]){

x=y;

y=parent[x];

}

return y

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

BTree\* BTree::successor(int val){

BTree\* np=this->search(val);

if(np){ //should be a node in the tree

if(np->right){ //node has a right subtree

return (np->right)->find\_Min();

}

else{ //go up the tree till you find a node which is a left child

BTree\* y= np->parent;

BTree\* x= np;

while( (y != NULL) && (y->right==x)){

x=y;

y=x->parent;

}

if(y==NULL)

return np; //np is Maximum element in the tree

else

return y;

}

}

else return NULL; //node with given int value is not in the tree

}

BTree\* BTree::predecessor(int val){

BTree\* np=this->search(val);

if(np){ //should be a node in the tree

if(np->left){ //node has a right subtree

return (np->left)->find\_Max();

}

else{ //go up the tree till you find a node which is a right child

BTree\* y= np->parent;

BTree\* x= np;

while( (y != NULL) && (y->left==x)){

x=y;

y=x->parent;

}

if(y==NULL)

return np; //np is Minimum element in the tree

else

return y;

}

}

else return NULL; //node with given int value is not in the tree

}

/\*\*\*Deletion Algorithm\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*\*\*\*\*Introduction to Algorithms* -*Second Ed.* Carmen/Leiserson/Rivest/Stein\*\*\*\*\*\*

delete(T, z):

//First determine node y to splice out

if left[z]==NULL or right[z]==NULL

y=z //node to splice out if z has only one child

else

y=successor(z) //node to splice out if z has two children

//set the non-NULL child of y to x, or set x to NULL if y has no children

if left[y] != NULL

x=left[y]

else

x=right[y]

//Now do the splicing

if x != NULL

parent[x]=parent[y]

if parent[y]==NULL

x=root[T]

else

if y==left[parent[y]]

left[parent[y]]=x

else

right[parent[y]]=x

//Splicing done

// if successor of z was the spliced out node, move y's key/data to z

if y != z

key[y]=key[z]

//copy non-key data too if present.

//return y if needed

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void BTree::remove(int val){ //delete node containing val as its key

BTree\* z=this->search(val); //get hold of a pointer to the node having val as its

key

BTree\* y,\*x;

if(z){ //should be a node in the tree

//y is the node to splice out, it is either the node with key=val, or its successor

if( (z->left == NULL)||(z->right ==NULL)) //z has at most one child

y=z;

else

y=this->successor(val); //z has two childern

//set x to be either the only child of y or NULL. Note that successor can't have two

children

if( y->left != NULL)

x=y->left;

else

x=y->right;

//###############Now do the splicing############

if (x != NULL){

x->parent=y->parent;

}

if(y->parent == NULL){

x=this->find\_root();

}

else if (y == (y->parent)->left){ //y is a left child of its parent

(y->parent)->left=x;

}

else { //y is a right child of its parent

(y->parent)->right=x;

}

//###################splicing done#############

// if successor of z was the spliced out node, move y's key/data to z

if (y != z)

z->key=y->key;

//copy non-key data too if present.

//return y if needed

}

else {

cout<<"Not Found\n";

}

}

int main(){

BTree\* bt=new BTree;

bt->insert(12);

bt->insert(5);

bt->insert(10);

bt->insert(21);

bt->insert(13);

bt->insert(3);

bt->insert(15);

bt->insert(22);

bt->insert(7);

cout<<"in-order->\t";bt->inOrderPrint();cout<<endl;

cout<<"pre-order->\t"; bt->preOrderPrint(); cout<<endl;

cout<<"post-order->\t"; bt->postOrderPrint(); cout<<endl;

cout<<endl;

cout<<"Max Value->"<<bt->find\_Max()->key<<endl;

cout<<"Min Value->"<<bt->find\_Min()->key<<endl;

cout<<"# of Nodes->"<<bt->countNodes()<<endl;

cout<<"# of Leaf Nodes->"<<bt->countLeafNodes()<<endl;

cout<<"Depth of the tree->"<<bt->find\_depth()<<endl;

cout <<"Successor of 10->"<< bt->successor(10)->key<<endl;

int m;

cout <<"Root is "<<(bt->find\_root())->key<<endl;

cout<<"Enter a value to remove\n";

cin>>m;

bt->remove(m);

cout<<"Deleted "<<m<<"\n";

cout<<"in-order->\t";bt->inOrderPrint();cout<<endl;

cout<<"pre-order->\t"; bt->preOrderPrint(); cout<<endl;

/\*

while (1){

cout<<"Successor/predecessor of which element do you need?\n";

cin>>m;

if(bt->search(m)){

cout <<"Successor of "<<m<<"->"<< bt->successor(m)->key<<endl;

cout <<"Predecessssor of "<<m<<"->"<< bt->predecessor(m)->key<<endl;

}

else

cout<<"Node not Found\n";

}

\*/

/\*

while (1){

cout<<"enter a value to serach\n";

cin>>m;

if(bt->search(m)){

cout<<"Found\n";

}else{

cout<<"Not Found\n";

}

}

\*/

}

**SESSION 6**

**Session 6:** Threaded binary tree- concepts, threading, insertion of nodes in in-order threaded binary tree

**Session Plan**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time (in min)** | **Content** | **Learning Aid/Methodology** | **Faculty Approach** | **Typical Student Activity** | **Skill /Competency Developed** |
| 05 | Introduction & revision | Presentation and Quiz | Introduces  Facilitates  monitors | Listens  Participate  Discuss | Knowledge |
| 20 | Threaded binary tree- concepts | Brain storming | Facilitates | Listens  Participate  Discuss | Comprehension  Application |
| 10 | threading, | Brain storming | Explain | Listen | Knowledge |
| 20 | insertion of nodes in in-order threaded binary tree | Brain storming  Quiz | Explain  Facilitates | Listens | Comprehension  Application |
| 5 | Conclusion and Summary | Key Words | List  Facilitates | Identifies | Knowledge  Comprehension |

**NOTES**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Threads:**

In linked representation of binary tree we can see that most of the nodes have Null values in their left and right pointer fields. It will be useful to use these pointer fields to keep some other information for operations in binary tree. Traversing is the most common operation in binary tree. We can use these pointer fields to contain the address pointer which points to the nodes higher in the tree. Such pointers which keeps the address of the nodes higher in the tree is called **thread.**

A binary tree which implements these pointers is called threaded binary tree.

We can have threading corresponding to any of the traversal.

There are 2 types of in-order threading,

1. One-way –

In this right field of the node will keep the thread pointer which will point to the next node in the sequence of the in-order traversal or we can say that right thread will point to the in-order successor of the node.

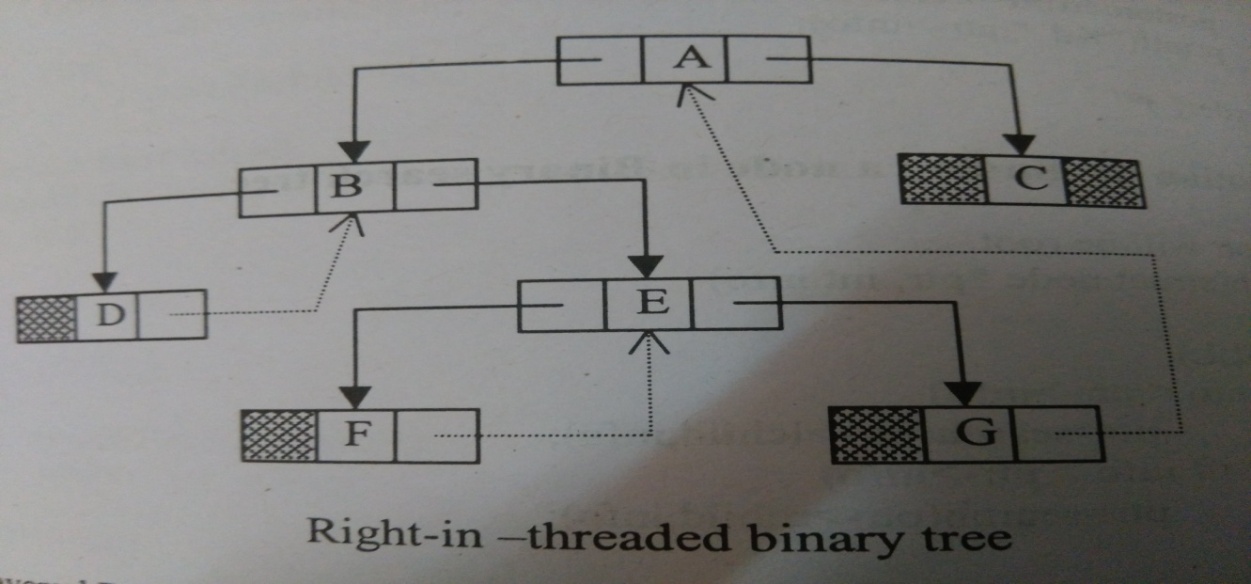
1. Two-way—

In this left thread will also point to in-order

Predecessor of the node along with right thread.

**Right In-threaded Binary Tree-**

If we use right field of node to take the thread then this is called right in-threaded binary tree.



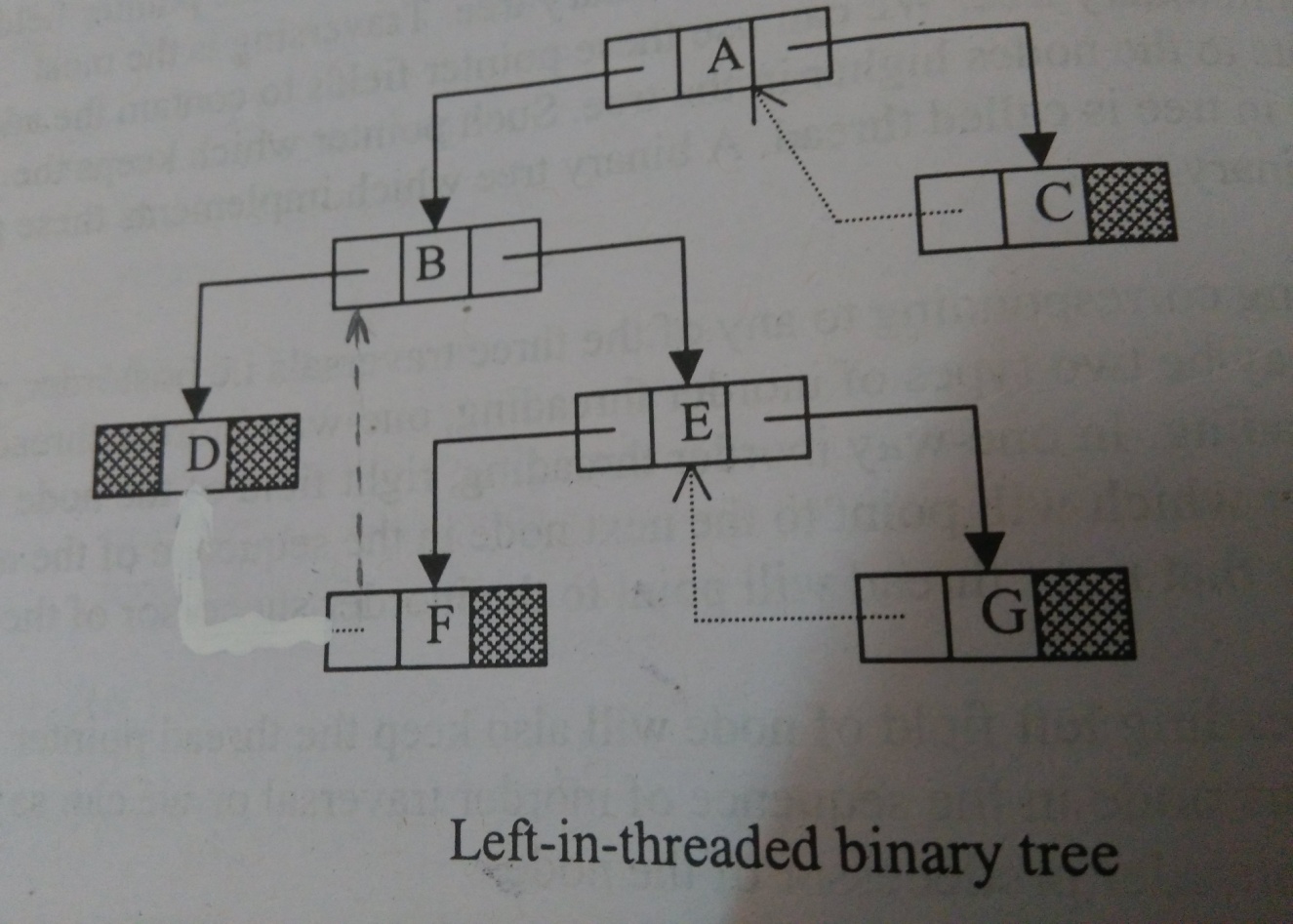
In-order Traversal – D B F E G A C

In this tree right child of node has threads which point to the next node in the sequence of the in-order traversal.

For e.g. D points to B

**Left In-threaded Binary tree-**

When we use left field of node to take the thread then this is called left in-threaded binary tree.

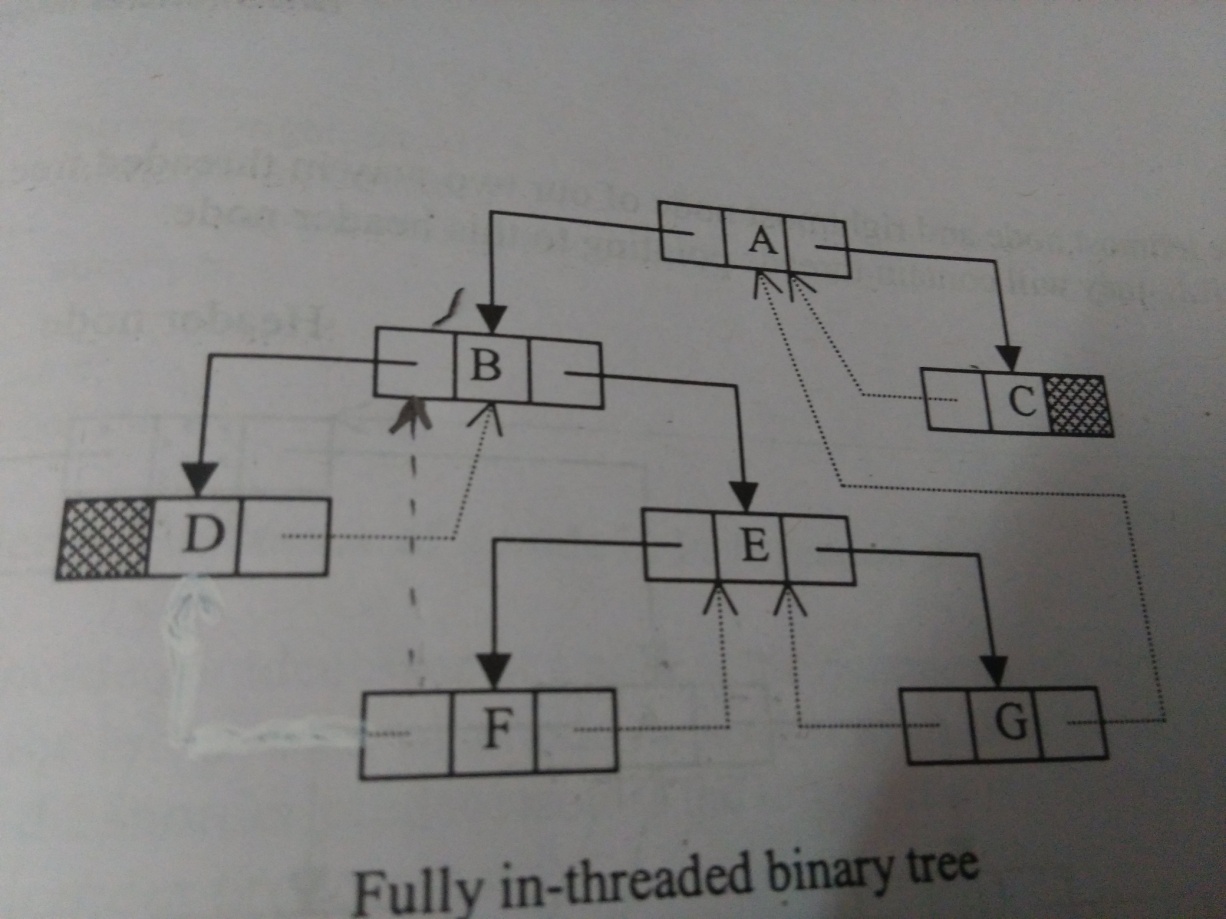


In this tree left child of node has threads which point to the previous node in the sequence of the in-order traversal.for e.g-

F points to B.

**Fully In-threaded Binary Tree-**

If both left and right fields are used for threading then this is called fully threaded or in-threaded binary tree.



Here left thread of F points to B which is in-order predecessor and right thread of F points to E which is in-order successor of F.

**Node structure for two ways In-threaded binary Tree**

Typedef enum {thread,link} Boolean;

struct node

{

struct node \*left\_ptr;

boolean left;

int info;

struct node \*right\_ptr;

boolean right;

}

Here we have taken two Boolean members left and right to differentiate between a thread and link.

left=link- pointer left\_ptr points to the left child of the node.

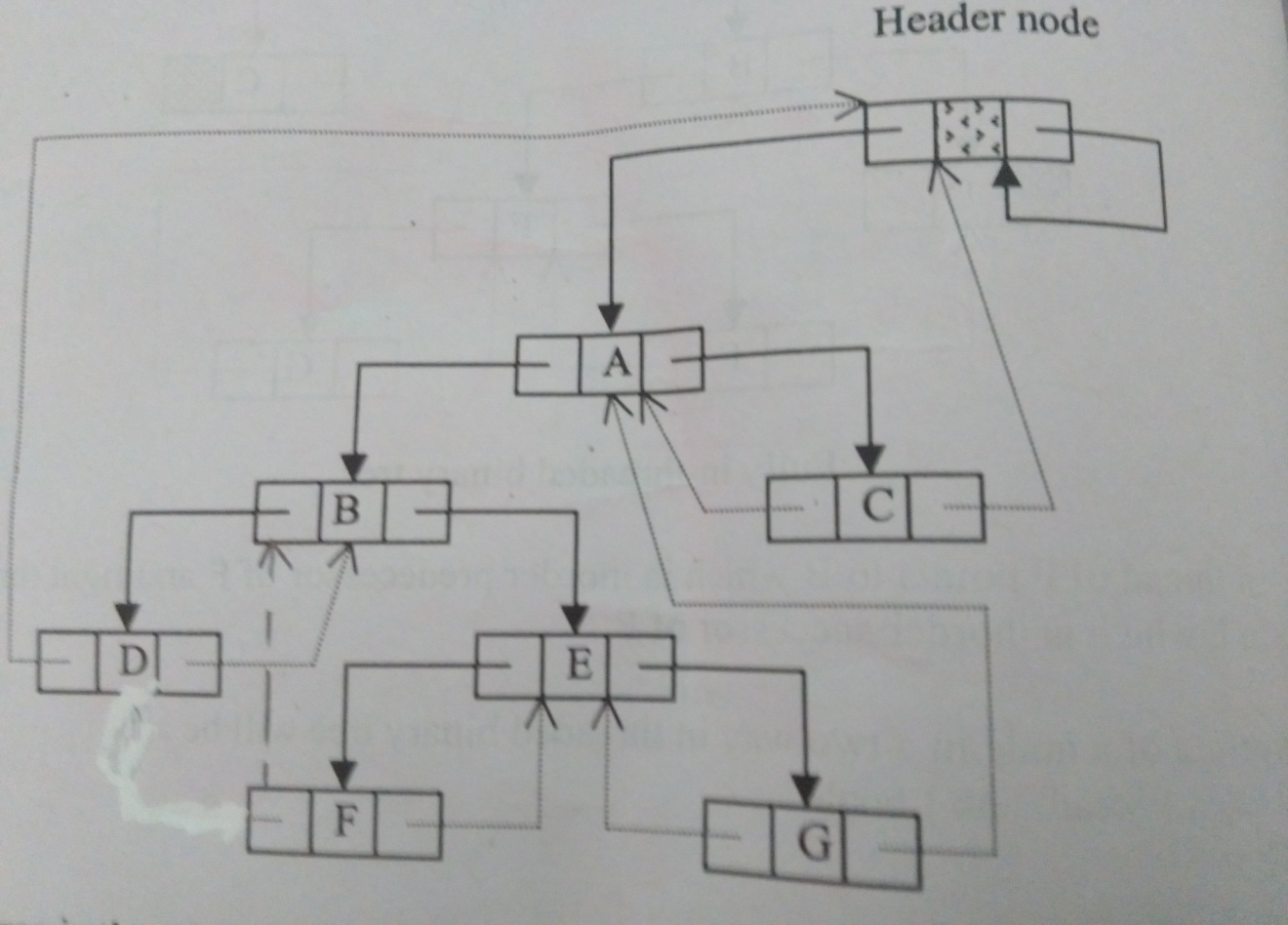
left=thread-pointer left\_ptr points to in-order predecessor of the node.

right=link- pointer right\_ptr points to the right child of the node.

right=thread-pointer right \_ptr points to in-order successor of the node.

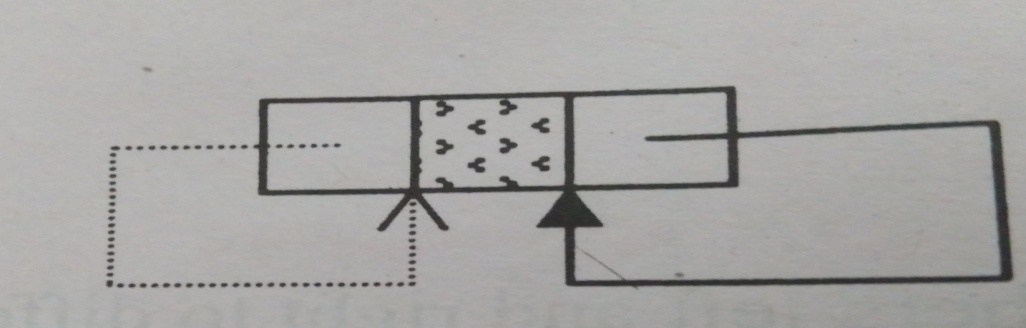
**TBT with Head node**

We have seen that if left or right pointer of a node is NULL then it is made a thread pointing to in-order predecessor or successor of the node. First node in in-order traversal has no predecessor and last node has no successor. So the left pointer of the leftmost node which is the first node in in-order traversal and the right pointer of the rightmost node which is the last node in in-order traversal contains NULL.So we can take a dummy node called the header node and we will represent our tree as the left subtree 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 node will be a thread pointing to itself.



So now the leftmost node and rightmost node of our two way in threaded tree will not contain NULL,they will contain threads pointing to this header node**.**

**An empty tree in threaded tree with header node can be represented as-**



So the condition for empty tree will be

head->left\_ptr=head;

**Finding in-order successor of a node in in-threaded tree**

We know that in-order successor of a node is the leftmost node in the right subtree of that node. So if the right pointer of a node consist of a link then we can traverse the right subtree and find the in-order successor. If the right pointer is a thread, then that thread will point to the in-order successor.

struct node \* in\_succ(struct node \*ptr)

{

struct node \*succ;

if(ptr->right==thread

succ=ptr->right\_ptr;

else

{

ptr =ptr->right\_ptr;

while(ptr->left==link)

ptr=ptr->left\_ptr;

succ=ptr;

}

return succ;

}

**Finding inorder predecessor of a node in in-threaded tree**

We know that in-order predecessor of a node is the rightmost node in the left subtree of that node. So if the left pointer of a node consists of a link then we can traverse the left subtree and find the in-order predecessor.If the left pointer is a thread ,then that thread will point to the in-order predecessor.

struct node \* in\_pred(struct node \*ptr)

{

struct node \*pred;

if(ptr->left==thread

pred=ptr->left\_ptr;

else

{

ptr =ptr->left\_ptr;

while(ptr->right==link)

ptr=ptr->right\_ptr;

pred=ptr;

}

return pred;

}

**In-order Traversal in in-threaded binary tree**

If the tree is right in-threaded then we can traverse it in in-order without use of stack or recursion. In in-order traversal the left most node is traversed first of all. So first we traverse the left most node of the tree and then with the help of in\_succ() function we find the in-order successor of each node and traverse it. We know that the rightmost node of the tree is the last node in in-order traversal and its right pointer is a thread pointing to header node, hence we will stop our process when we reach header node.

inorder()

{

struct node \* ptr;

if(head->left \_ptr==head)

{

printf(“Tree is Empty”);

return;

}

ptr=head->left;

//find the left most node

while(ptr->left==link)

ptr=ptr->left\_ptr;

printf(“%d”,ptr->info) ;

while(1)

{

ptr=in\_succ(ptr);

if(ptr==head)

break;

printf(“%d”,ptr->info) ;

}

}//end of while

}//end of inorder function

Threaded Binary Tree

[Inorder traversal of a Binary tree](http://courses.geeksforgeeks.org/section/193) is either done using recursion or [with the use of a auxiliary stack](http://courses.geeksforgeeks.org/section/198). The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL pointer to the inorder successor of the node (if it exists).

There are two types of threaded binary trees.  
**Single Threaded:** Where a NULL right pointers is made to point to the inorder successor (if successor exists)

Single Left Threaded: - Figure shows left null pointers to point to the predecessor of

that node in inorder traversal. This can be used for reverse of inorder traversal i.e. Right

Visit Left for any subtree. i.e. first go right, then visit root and go to left of the root.



Single Right Threaded: - Figure 1.4 shows right null pointers to point to the successor of

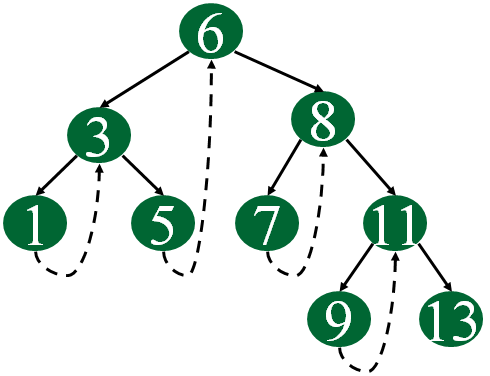
that node in inorder traversal. This is also called right-in threaded trees

**Double Threaded:** Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.



The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



**C representation of a Threaded Node**   
Following is C representation of a single threaded node.

struct Node

{

int data;

Node \*left, \*right;

bool rightThread;

}

Since right pointer is used for two purposes, the boolean variable rightThread is used to indicate whether right pointer points to right child or inorder successor. Similarly, we can add leftThread for a double threaded binary tree.

Following is C code for inorder traversal in a threaded binary tree.

// Utility function to find leftmost node in a tree rooted with n

struct Node\* leftMost(struct Node \*n)

{

if (n == NULL)

return NULL;

while (n->left != NULL)

n = n->left;

return n;

}

// C code to do inorder traversal in a threadded binary tree

void inOrder(struct Node \*root)

{

struct Node \*cur = leftmost(root);

while (cur != NULL)

{

printf("%d ", cur->data);

// If this node is a thread node, then go to

// inorder successor

if (cur->rightThread)

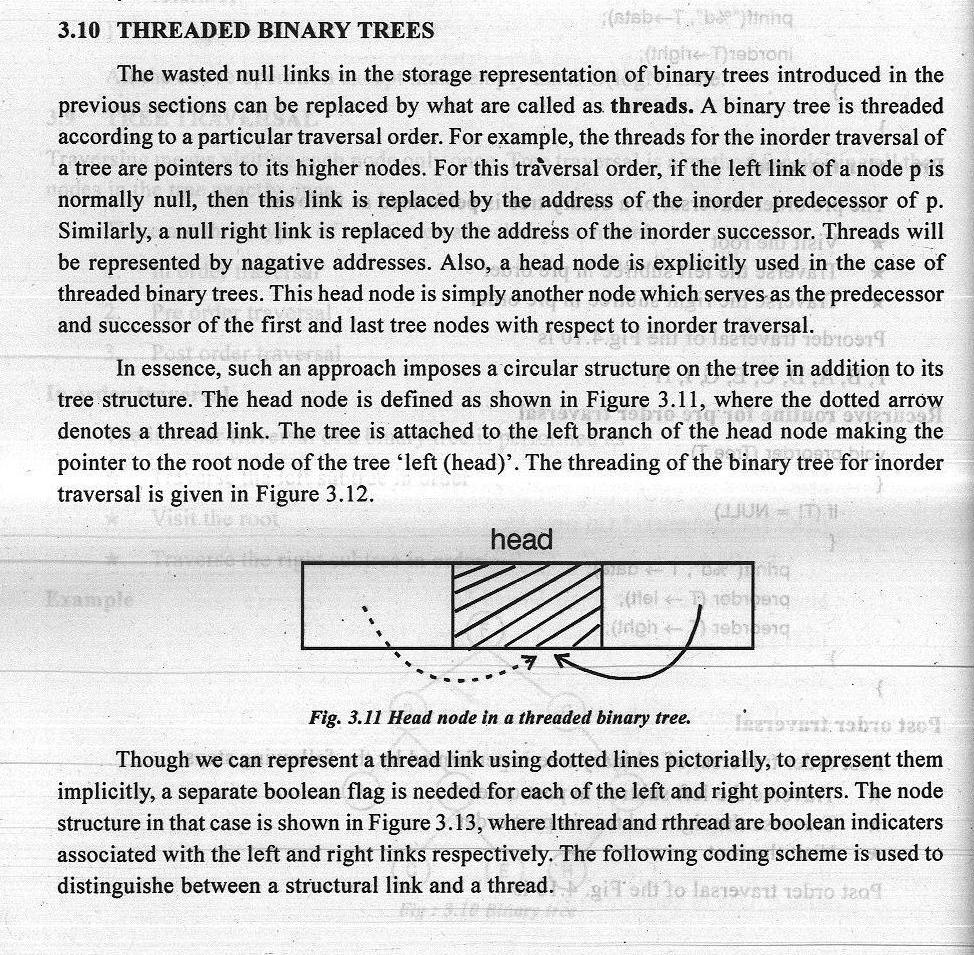
cur = cur->right;

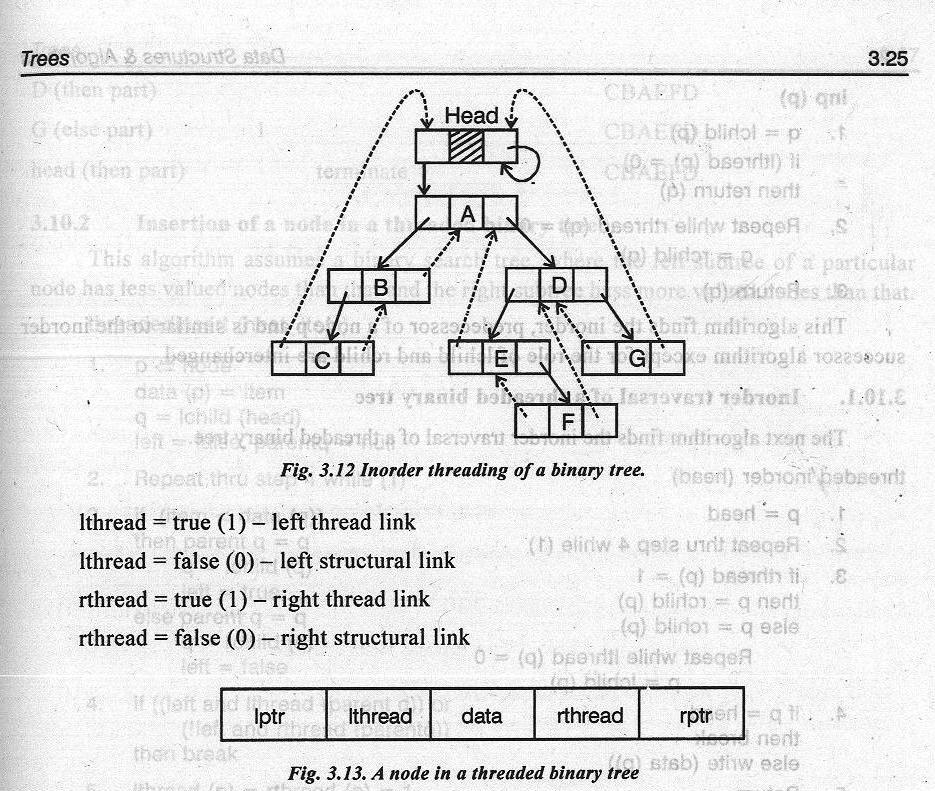
else // Else go to the leftmost child in right subtree

cur = leftmost(cur->right);

}

}

****



The node structure for a threaded binary tree varies a bit and its like this --

struct node

{

int data;

struct node \*lptr, \*rptr, \*lthread, \*rthread;

}

Let's make the Threaded Binary tree out of a normal binary tree...



The INORDER traversal for the above tree is -- D B A E C.

So, the respective Threaded Binary tree will be --



B has no right child and its inorder successor is A and so a thread has been made in

between them. Similarly, for D and E. C has no right child but it has no inorder successor even, so it has a hanging thread.

**Non recursive Inorder traversal for a Threaded Binary Tree**

As this is a non-recursive method for traversal, it has to be an iterative procedure;

meaning, all the steps for the traversal of a node have to be under a loop so that the same can be applied to all the nodes in the tree.

I'll consider the INORDER traversal again. Here, for every node, we'll visit the left subtree (if it exists) first (if and only if we haven't visited it earlier); then we visit (i.e print its value, in our case) the node itself and then the right sub-tree (if it exists). If the right subtree is not there, we check for the threaded link and make the threaded node the current node in consideration.

Algorithm:-

Step-1: For the current node check whether it has a left child which is not there in

the visited list. If it has then go to step-2 or else step-3.

Step-2: Put that left child in the list of visited nodes and make it your current node

in consideration. Go to step-6.

Step-3: For the current node check whether it has a right child. If it has then go to

step-4 else go to step-5

Step-4: Make that right child as your current node in consideration. Go to step-6.

Step-5: Check for the threaded node and if its there make it your current node.

Step-6: Go to step-1 if all the nodes are not over otherwise quit

Please, follow the example given below.



List of visited nodes INORDER

step-1:'A' has a left child i.e B, which has

not been visited. So, we put B in our B

"list of visited nodes" and B becomes our

current node inconsideration.

step-2:'B' also has a left child, 'D', which is

not there in our list of visited nodes.

So, we put 'D' in that list and make it B D

our current node in consideration.

step-3:'D' has no left child, so we print 'D'.

Then we check for its right child. 'D'

has no right child and thus we check B D D

for its thread-link. It has a thread

going till node 'B'. So, we make 'B'

as our current node in consideration.

step-4:'B' certainly has a left child but its

already in our list of visited nodes.

So, we print 'B'. Then we check for B DA D B

its right child but it doesn't exist. So,

we make its threaded node (i.e 'A')

as our current node in consideration.

step-5:'A' has a left child, 'B', but its

already there in the list of visited

nodes. So, we print 'A'. Then we B DA C D BA

check for its right child. 'A' has a

right child, 'C' and its not there in

our list of visited nodes. So, we add

it to that list and we make it our

current node in consideration.

step-6:'C' has 'E' as the left child and its not

there in our list of visited nodes

even. So, we add it to that list and B DAC E D B A

make it our current node in

consideration. and finally..... D B AE C

**Algorithm Implementation for inorder threaded binary tree:**

struct NODE

{

struct NODE \*left;

int value;

struct NODE \*right;

struct NODE \*thread;

}

inorder(struct NODE \*curr)

{

while(all the nodes are not over )

{

if(curr->left != NULL && ! visited(curr->left))

{

visit(curr->left);

curr = curr->left;

}

else

{

**display the current value; // instead using🡪 printf(“%d”,curr->value)**

if(curr->right != NULL)

curr = curr->right;

else

if(curr->thread != NULL)

curr = curr->thread;

}

}

}

Note:-

The functions - visit( ) maintains a linked list of already visited nodes, visited( )

returns a TRUE value if it finds a particular node, in the list maintained by visit( ),

otherwise FALSE.

**Traversal of TBT/TBST**

* FOLLOWING METHODS ARE EQUALLY APPLICABLE TO BOTH TBT AND TBST
* Preorder
* Inorder
* postorder

**Preorder successor**

tbtnode\* presucc(tbtnode \*t)

{

if(t->lbit==1)//normal left child

return(t->lchild);

if(t->rbit==1)//normal right child

return(t->rchild);

while(t->rbit==0) //if thread then go up

t=t->rchild;

return(t->rchild);

}

**Preorder traversal**

void tpreorder(tbtnode \*t)

{

tbtnode\* h;

h=t; //assuming t is head initially

t=t->lchild;

while(t != h)

{

cout<<“ ”<<t->data;

t=presucc(t);

}

}

**Inorder successor**

tbtnode\* insucc(tbtnode \*t)

{

if(t->rbit == 0)//right child is thread link

return(t->rchild);

t=t->rchild;

while(t->lbit == 1) //leftmost child of right subtree

t=t->lchild;

return(t);

}

**Inorder traversal**

void tinorder(tbtnode \*t)

{

tbtnode\* h;

h=t; //assuming t is head initially

t=t->lchild;

while(t->lbit == 1)

t=t->lchild;

while(t != h)

{

cout<<“ ”<<t->data;

t=insucc(t);

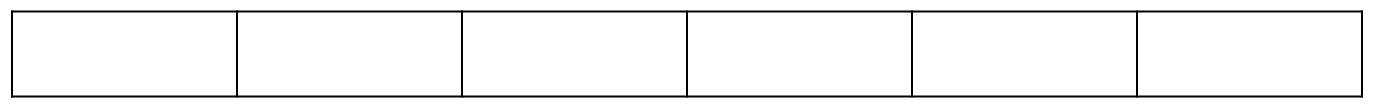
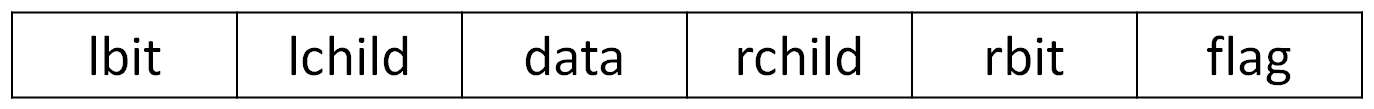
}

}

**Postorder traversal hints**

* Postorder is bit complex than earlier 2
* In postorder, parent is printed/visited after both of its children
* Here, node structure is modified as we need to locate parent of each node.
* Node structure needs to altered with 1 additional field called as flag (on next slide):
  + int flag;
  + if flag=0 node is left child of its parent
  + if flag=1 node is right child of its parent

**Changed node structure for postorder**



class tbtnode

{

int data;

tbtnode \*lchild, \*rchild;

int lbit, rbit;

int flag; // if flag=0 node is left child of its parent else rt

};

**To find parent of any node t**

Void parent(tbtnode \*t)

{

if(t->flag==0) // t is left child of some node

{

while(t->rbit==1)

t=t->rchild;

return(t->rchild);

}

else // t is right child of some node

{

while(t->lbit==1)

t=t->lchild;

return(t->lchild);

}

}

**Postorder successor**

tbtnode\* postsucc(tbtnode \*t)

{

if(t->flag==0) // t is left child

{

t=parent(t);

if(t->rbit==0) //parent t has no rchild

return(t);

else //parent t has rchild

{

t=t->rchild;

while(t->lbit==1 || t->rbit==1)

if(t->lbit==1 )

t=t->lchild;

else

t=t->rchild;

return(t);

}

}

else

return (parent (t) );

}

**Postorder traversal**

Void tpostorder(tbtnode \*t)

{

tbtnode\* h;

h=t; //assuming t is head initially

t=t->lchild;

while(t->lbit==1 || t->rbit==1)

if(t->lbit==1 )

t=t->lchild;

else

t=t->rchild;

while(t != h)

{

cout<<“ ”<<t->data;

t=postsucc(t);

}

}

**Advantages of Threaded Binary Trees**

A threaded binary tree makes it possible to traverse the values in the binary tree

via a linear traversal that is more rapid than a recursive in-order traversal or

iterative one which uses stacks.

It is also possible to discover the parent of a node from a threaded binary tree,

without explicit use of parent pointers or a stack. This can be useful where stack

space is limited, or where a stack of parent pointers is unavailable (for finding the

parent pointer via DFS).This is possible, because if a node (k) has a right child (m)

then m's left pointer must be either a child, or a thread back to k. In the case of a

left child, that left child must also have a left child or a thread back to k, and so we

can follow m's left children until we find a thread, pointing back to k. The situation

is similar for when m is the left child of k.

Any node can be accessed from any other node. Threads are usually more to

upward whereas links are downward. Thus in a threaded tree, one can move in

either direction and nodes are in fact circularly linked. This is not possible in

unthreaded counter part because there we can move only in downward direction

starting form root.

Drawbacks of Threaded Binary Trees

1. Slower tree creation and updating since threads need to be maintained.

2. In theory, threaded trees need two extra bits per node to indicate whether each

child pointer points to an ordinary node or the node's successor/predecessor node.

3. Insertion into and deletions from a threaded tree are all although time consuming

**Applications Of Threaded Trees**

Threaded trees are a important data structure where trees are created once with

very less of insertions and deletions operations there on and more of traversal

operations (specifically depth first traversals) are more in number. In such

situations the threaded trees act as a boon for the system performance by reducing

the space required for stacks.

It is also possible to discover the parent of a node from a threaded binary tree,

without explicit use of parent pointers or a stack, albeit slowly. This can be useful

where stack space is limited, or where a stack of parent pointers is unavailable (for

finding the parent pointer via DFS).

This is possible, because if a node (k) has a right child (m) then m's left pointer must be

either a child, or a thread back to k. In the case of a left child, that left child must also

have a left child or a thread back to k, and so we can follow m's left children until we find

a thread, pointing back to k. The situation is similar for when m is the left child of k.

**Applications Of Threaded Trees**

Threaded trees are a important data structure where trees are created once with

very less of insertions and deletions operations there on and more of traversal

operations (specifically depth first traversals) are more in number. In such

situations the threaded trees act as a boon for the system performance by reducing

the space required for stacks.

It is also possible to discover the parent of a node from a threaded binary tree,

without explicit use of parent pointers or a stack, albeit slowly. This can be useful

where stack space is limited, or where a stack of parent pointers is unavailable (for

finding the parent pointer via DFS).

This is possible, because if a node (k) has a right child (m) then m's left pointer must be

either a child, or a thread back to k. In the case of a left child, that left child must also

have a left child or a thread back to k, and so we can follow m's left children until we find

a thread, pointing back to k. The situation is similar for when m is the left child of k.

**SESSION 7**

**SESSION 7:** Threaded binary tree- deletion of nodes in in-order threaded binary tree

**Session Plan**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

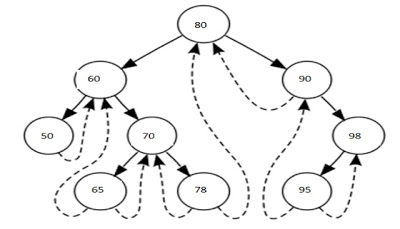
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time (in min)** | **Content** | **Learning Aid/Methodology** | **Faculty Approach** | **Typical Student Activity** | **Skill /Competency Developed** |
| 05 | Introduction | Presentation and Quiz | Introduces  Facilitates  monitors | Listens  Participate  Discuss | Knowledge |
| 15 | Threaded binary tree | Brain storming | Facilitates | Listens  Participate  Discuss | Comprehension  Application |
| 15 | deletion of nodes | Brain storming | Explain | Listen | Knowledge |
| 20 | in-order threaded binary tree | Brain storming  Quiz | Explain  Facilitates | Listens | Comprehension  Application |
| 5 | Conclusion and Summary | Key Words | List  Facilitates | Identifies | Knowledge  Comprehension |

**NOTES**

### C Program To Perform Insertion, Deletion & Traversal In Threaded Binary Search Tree

* The left and right child pointers in binary search tree are NULL.
* But in threaded binary search tree, left child pointer will point to the predecessor and the right child will point to the successor of the current node.
* For traversal in Binary search tree, we need to keep track of list of the nodes present above the current node.  It leads to additional usage of space and time.  But, this could be avoided by using Threaded Binary Search Tree.

**Example for Threaded Binary Search Tree:**

[](http://4.bp.blogspot.com/-520HfO5w3CA/UYOobT_n2nI/AAAAAAAAACw/cMUXvc6fC2A/s1600/Threaded+Binary+Tree.jpg)

#include <stdio.h>  
  #include <stdlib.h>  
  
  enum marker {  
        CHILD,  
        THREAD  
  };  
  
  struct tbstNode {  
        int data;  
        struct tbstNode \*link[2];  
        int marker[2];  
  };  
  
  struct tbstNode \*root = NULL;  
  
  struct tbstNode \* createNode (int data) {  
        struct tbstNode \*newNode;  
        newNode = (struct tbstNode \*)malloc(sizeof (struct tbstNode));  
        newNode->data = data;  
        newNode->link[0] = newNode->link[1] = NULL;  
        newNode->marker[0] = newNode->marker[1] = THREAD;  
        return newNode;  
  }

  void insertion(int data) {

        struct tbstNode \*parent, \*newNode, \*temp;

        int path;

        if (!root) {

                root = createNode(data);

                return;

        }

        parent = root;

        /\* find the location to insert the new node \*/

        while (1) {

                if (data == parent->data) {

                        printf("Duplicates Not Allowed\n");

                        return;

                }

                path = (data > parent->data) ? 1 : 0;

                if (parent->marker[path] == THREAD)

                        break;

                else

                        parent = parent->link[path];

        }

        /\*

         \* newnode's left points to predecessor and

         \* right to successor

         \*/

        newNode = createNode(data);

        newNode->link[path] = parent->link[path];

        parent->marker[path] = CHILD;

        newNode->link[!path] = parent;

        parent->link[path] = newNode;

        return;

  }

  void delete(int data) {

        struct tbstNode \*current, \*parent, \*temp;

        int path;

        parent = root;

        current = root;

        /\* search the node to delete \*/

        while (1) {

                if (data == current->data)

                        break;

                path = (data > current->data) ? 1 : 0;

                if (current->marker[path] == THREAD) {

                        printf("Given data is not available!!\n");

                        return;

                }

                parent = current;

                current = current->link[path];

        }

        if (current->marker[1] == THREAD) {

                if (current->marker[0] == CHILD) {

                        /\* node with single child \*/

                        temp = current->link[0];

                        while (temp->marker[1] == CHILD) {

                                temp = temp->link[1];

                        }

                        temp->link[1] = current->link[1];

                        if (current == root) {

                                root = current->link[0];

                        } else {

                                parent->link[path] = current->link[0];

                        }

                } else {

                        /\* deleting leaf node \*/

                        if (current == root) {

                                root = NULL;

                        } else {

                                parent->link[path] = current->link[path];

                                parent->marker[path] = THREAD;

                        }

                }

        } else {

                temp = current->link[1];

                /\*

                 \* node with two child - whose right child has

                 \* no left child

                 \*/

                if (temp->marker[0] == THREAD) {

                        temp->link[0] = current->link[0];

                        temp->marker[0] = current->marker[0];

                        if (temp->marker[0] == CHILD) {

                                struct tbstNode \*x = temp->link[0];

                                while (x->marker[1] == CHILD) {

                                        x = x->link[1];

                                }

                                x->link[1] = temp;

                        }

                        if (current == root) {

                                root = temp;

                        } else {

                                printf("path: %d data:%d\n", path, parent->data);

                                parent->link[path] = temp;

                        }

                } else {

                        /\* node with two child \*/

                        struct tbstNode \*child;

                        while (1) {

                                child  = temp->link[0];

                                if (child->marker[0] == THREAD)

                                        break;

                                temp = child;

                        }

                        if (child->marker[1] == CHILD)

                                temp->link[0] = child->link[1];

                        else {

                                temp->link[0] = child;

                                temp->marker[0] = THREAD;

                        }

                        child->link[0] = current->link[0];

                        /\* update the links \*/

                        if (current->marker[0] == CHILD) {

                                struct tbstNode \*x = current->link[0];

                                while(x->marker[1] == CHILD)

                                        x = x->link[1];

                                x->link[1] = child;

                                child->marker[0] = CHILD;

                        }

                        child->link[1] = current->link[1];

                        child->marker[1] = CHILD;

                        if (current == root)

                                root = child;

                        else

                                parent->link[path] = child;

                }

        }

        /\* deallocation \*/

        free(current);

        return;

  }

  void traversal() {

        struct tbstNode \*myNode;

        if (!root) {

                printf("Threaded Binary Tree Not Exists!!\n");

                return;

        }

        myNode = root;

        while (1) {

                while(myNode->marker[0] == CHILD) {

                        myNode = myNode->link[0];

                }

                printf("%d ", myNode->data);

                myNode = myNode->link[1];

                if (myNode) {

                        printf("%d ", myNode->data);

                        myNode = myNode->link[1];

                }

                if (!myNode)

                        break;

        }

        printf("\n");

        return;

  }

  void search(int data) {

        struct tbstNode \*myNode;

        int path;

        if (!root) {

                printf("Tree Not Available!!\n");

                return;

        }

        myNode = root;

        while (1) {

                if (myNode->data == data) {

                        printf("Given data present in TBST!!\n");

                        return;

                }

                path = (data > myNode->data) ? 1 : 0;

                if (myNode->marker[path] == THREAD)

                        break;

                else

                        myNode = myNode->link[path];

        }

        printf("Given data is not present in TBST!!\n");

        return;

  }

  int main () {

        int data, ch;

        while (1) {

                printf("1. Insertion\t2. Deletion\n");

                printf("3. Searching\t4. Traversal\n");

                printf("5. Exit\nEnter your choice:");

                scanf("%d", &ch);

                switch (ch) {

                        case 1:

                                printf("Enter your input data:");

                                scanf("%d", &data);

                                insertion(data);

                                break;

                        case 2:

                                printf("Enter your input data:");

                                scanf("%d", &data);

                                delete(data);

                                break;

                        case 3:

                                printf("Enter your input data:");

                                scanf("%d", &data);

                                search(data);

                                break;

                        case 4:

                                traversal();

                                break;

                        case 5:

                                exit(0);

                        default:

                                printf("You have entered wrong option!!\n");

                                break;

                }

                printf("\n");

        }

  }

**SESSION 8**

**SESSION 8:** Threaded binary tree- deletion of nodes in in-order threaded binary tree

**Session Plan**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time (in min)** | **Content** | **Learning Aid/Methodology** | **Faculty Approach** | **Typical Student Activity** | **Skill /Competency Developed** |
| 05 | Introduction | Presentation and Quiz | Introduces  Facilitates  monitors | Listens  Participate  Discuss | Knowledge |
| 15 | Threaded binary tree | Brain storming | Facilitates | Listens  Participate  Discuss | Comprehension  Application |
| 15 | deletion of nodes continue | Brain storming | Explain | Listen | Knowledge |
| 20 | in-order threaded binary tree | Brain storming  Quiz | Explain  Facilitates | Listens | Comprehension  Application |
| 5 | Conclusion and Summary | Key Words | List  Facilitates | Identifies | Knowledge  Comprehension |

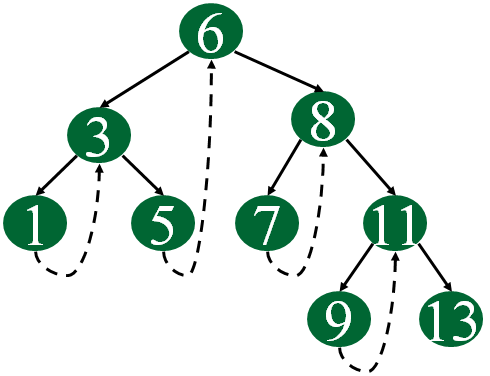
**NOTES**

[Inorder traversal of a Binary tree](http://www.geeksforgeeks.org/618/) is either be done using recursion or [with the use of a auxiliary stack](http://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/). The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

There are two types of threaded binary trees.  
***Single Threaded:***Where a NULL right pointers is made to point to the inorder successor (if successor exists)

***Double Threaded:*** Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.  
[](http://quiz.geeksforgeeks.org/wp-content/uploads/2014/07/threadedBT.png)

**C representation of a Threaded Node**  
Following is C representation of a single threaded node.

|  |
| --- |
| struct Node  {      int data;      Node \*left, \*right;      bool rightThread;  } |

Run on IDE

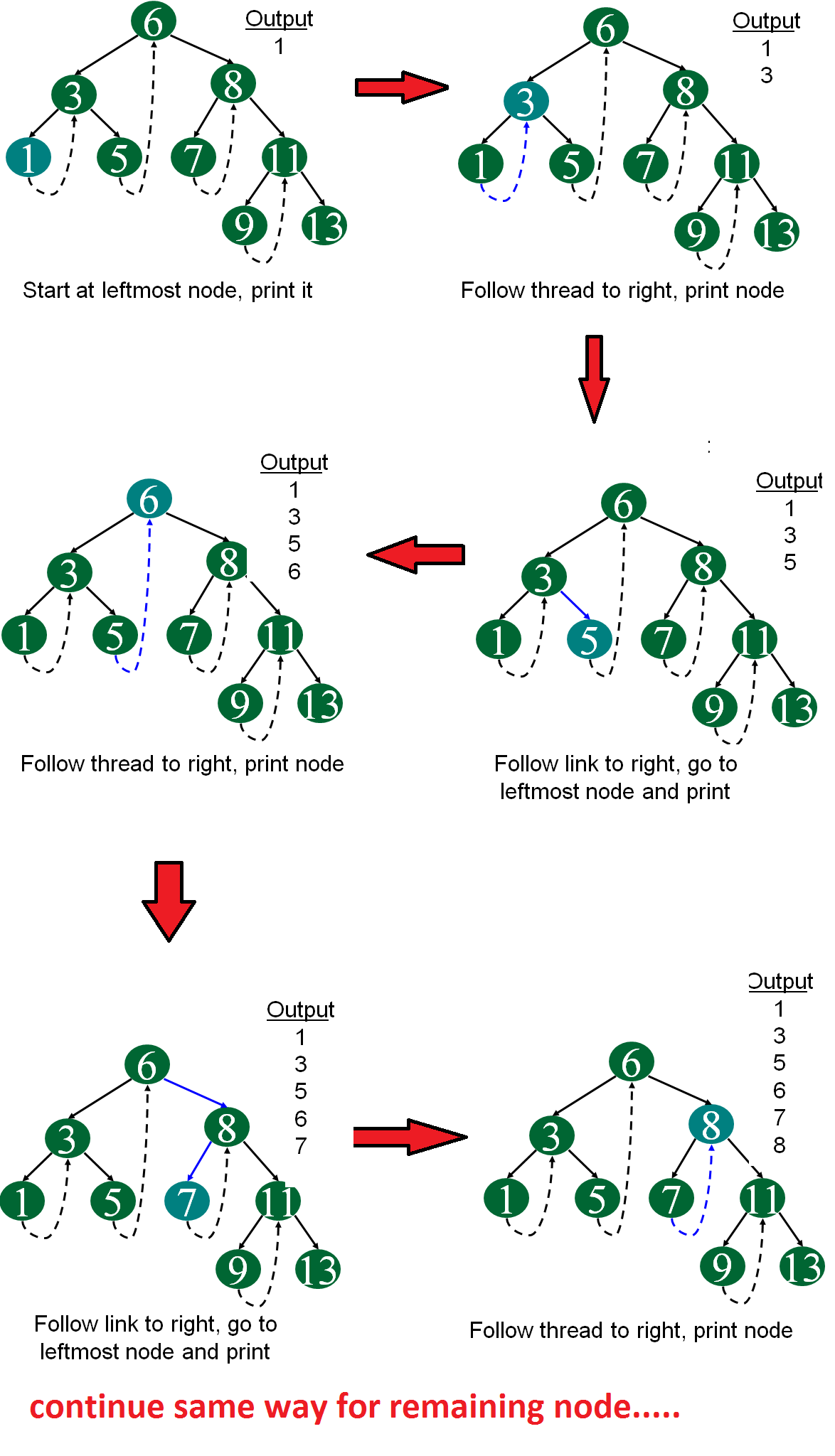
Since right pointer is used for two purposes, the boolean variable rightThread is used to indicate whether right pointer points to right child or inorder successor. Similarly, we can add leftThread for a double threaded binary tree.

**Inorder Taversal using Threads**  
Following is C code for inorder traversal in a threaded binary tree.

|  |
| --- |
| // Utility function to find leftmost node in atree rooted with n  struct Node\* leftMost(struct Node \*n)  {      if (n == NULL)         return NULL;        while (n->left != NULL)          n = n->left;        return n;  }    // C code to do inorder traversal in a threadded binary tree  void inOrder(struct Node \*root)  {      struct Node \*cur = leftmost(root);      while (cur != NULL)      {          printf("%d ", cur->data);            // If this node is a thread node, then go to          // inorder successor          if (cur->rightThread)              cur = cur->right;          else // Else go to the leftmost child in right subtree              cur = leftmost(cur->right);      }  } |

Run on IDE

Following diagram demonstrates inorder order traversal using threads.

[](http://quiz.geeksforgeeks.org/wp-content/uploads/2014/07/threadedTraversal.png)

We will soon be discussing insertion and deletion in threaded binary trees.

**SESSION 9**

**SESSION 9:** **Case Study**- Use of binary tree in expression tree-evaluation and Huffman's coding

**Session Plan**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time (in min)** | **Content** | **Learning Aid/Methodology** | **Faculty Approach** | **Typical Student Activity** | **Skill /Competency Developed** |
| 05 | Introduction | Presentation and Quiz | Introduces  Facilitates  monitors | Listens  Participate  Discuss | Knowledge |
| 25 | Use of binary tree in expression tree-evaluation | Brain storming | Facilitates | Listens  Participate  Discuss | Comprehension  Application |
| 25 | Huffman's coding | Brain storming | Explain | Listen | Knowledge |
| 5 | Conclusion and Summary | Key Words | List  Facilitates | Identifies | Knowledge  Comprehension |

**NOTE**

**Huffman Coding**

Huffman coding is based on the frequency of occurance of a data item (pixel in images). The principle is to use a lower number of bits to encode the data that occurs more frequently. Codes are stored in a *Code Book* which may be constructed for each image or a set of images. In all cases the code book plus encoded data must be transmitted to enable decoding.

The Huffman algorithm is now briefly summarised:

* A bottom-up approach

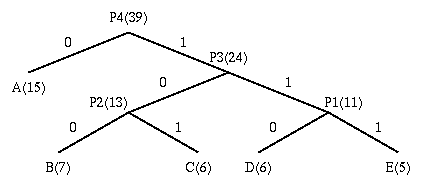
1. Initialization: Put all nodes in an OPEN list, keep it sorted at all times (e.g., ABCDE).

2. Repeat until the OPEN list has only one node left:

(a) From OPEN pick two nodes having the lowest frequencies/probabilities, create a parent node of them.

(b) Assign the sum of the children's frequencies/probabilities to the parent node and insert it into OPEN.

(c) Assign code 0, 1 to the two branches of the tree, and delete the children from OPEN.



Symbol Count log(1/p) Code Subtotal (# of bits)

------ ----- -------- --------- --------------------

A 15 1.38 0 15

B 7 2.48 100 21

C 6 2.70 101 18

D 6 2.70 110 18

E 5 2.96 111 15

TOTAL (# of bits): 87

The following points are worth noting about the above algorithm:

* Decoding for the above two algorithms is trivial as long as the coding table (the statistics) is sent before the data. (There is a bit overhead for sending this, negligible if the data file is big.)
* **Unique Prefix Property**: no code is a prefix to any other code (all symbols are at the leaf nodes) -> great for decoder, unambiguous.
* If prior statistics are available and accurate, then Huffman coding is very good.

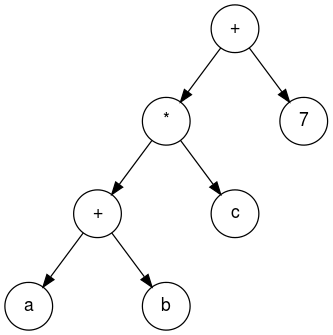
In the above example:

Number of bits needed for Huffman Coding is: 87 / 39 = 2.23

**Expression Tree**

A **binary expression tree** is a specific kind of a [binary tree](https://en.wikipedia.org/wiki/Binary_tree) used to represent expressions. Two common types of expressions that a binary expression tree can represent are [algebraic](https://en.wikipedia.org/wiki/Algebra)[[1]](https://en.wikipedia.org/wiki/Binary_expression_tree#cite_note-brpreiss-1) and [boolean](https://en.wikipedia.org/wiki/Boolean_algebra" \o "Boolean algebra). These trees can represent expressions that contain both [unary](https://en.wikipedia.org/wiki/Unary_operation) and [binary](https://en.wikipedia.org/wiki/Binary_function) operators.[[1]](https://en.wikipedia.org/wiki/Binary_expression_tree#cite_note-brpreiss-1)

Each node of a binary tree, and hence of a binary expression tree, has zero, one, or two children. This restricted structure simplifies the processing of expression trees.



The leaves of a binary expression tree are operands, such as constants or variable names, and the other nodes contain operators. These particular trees happen to be binary, because all of the operations are binary, and although this is the simplest case, it is possible for nodes to have more than two children. It is also possible for a node to have only one child, as is the case with the unary minus operator. An expression tree, *T*, can be evaluated by applying the operator at the root to the values obtained by recursively evaluating the left and right subtrees.

### Traversal

An algebraic expression can be produced from a binary expression tree by recursively producing a parenthesized left expression, then printing out the operator at the root, and finally recursively producing a parenthesized right expression. This general strategy (left, node, right) is known as an [in-order traversal](https://en.wikipedia.org/wiki/Tree_traversal). An alternate traversal strategy is to recursively print out the left subtree, the right subtree, and then the operator. This traversal strategy is generally known as [post-order traversal](https://en.wikipedia.org/wiki/Tree_traversal). A third strategy is to print out the operator first and then recursively print out the left and right subtree.[[2]](https://en.wikipedia.org/wiki/Binary_expression_tree#cite_note-Gopal2010-2)

These three standard depth-first traversals are representations of the three different expression formats: infix, postfix, and prefix. An infix expression is produced by the inorder traversal, a postfix expression is produced by the post-order traversal, and a prefix expression is produced by the pre-order traversal.

#### Infix traversal

When an infix expression is printed, an opening and closing parenthesis must be added at the beginning and ending of each expression. As every subtree represents a subexpression, an opening parenthesis is printed at its start and the closing parenthesis is printed after processing all of its children.

Pseudocode:

Algorithm infix (tree)

*/\*Print the infix expression for an expression tree.*

*Pre: tree is a pointer to an expression tree*

*Post: the infix expression has been printed\*/*

**if** (tree not empty)

**if** (tree token is operator)

print (open parenthesis)

end **if**

infix (tree left subtree)

print (tree token)

infix (tree right subtree)

**if** (tree token is operator)

print (close parenthesis)

end **if**

end **if**

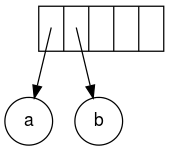
end infix

Construction of an expression tree

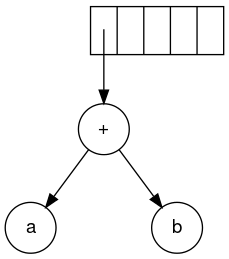
The evaluation of the tree takes place by reading the postfix expression one symbol at a time. If the symbol is an operand, one-node tree is created and a pointer is pushed onto a [stack](https://en.wikipedia.org/wiki/Stack_(abstract_data_type)). If the symbol is an operator, the pointers are popped to two trees *T1* and *T2* from the stack and a new tree whose root is the operator and whose left and right children point to *T2* and *T1* respectively is formed . A pointer to this new tree is then pushed to the Stack.

### Example

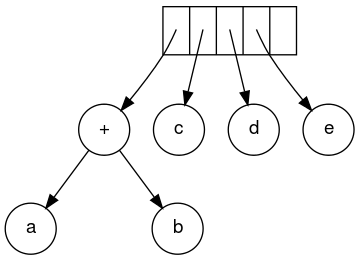
The input is: a b + c d e + \* \* Since the first two symbols are operands, one-node trees are created and pointers are pushed to them onto a stack. For convenience the stack will grow from left to right.



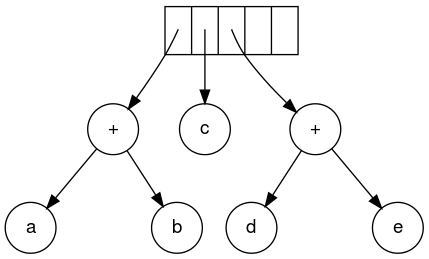
The next symbol is a '+'. It pops the two pointers to the trees, a new tree is formed, and a pointer to it is pushed onto to the stack.



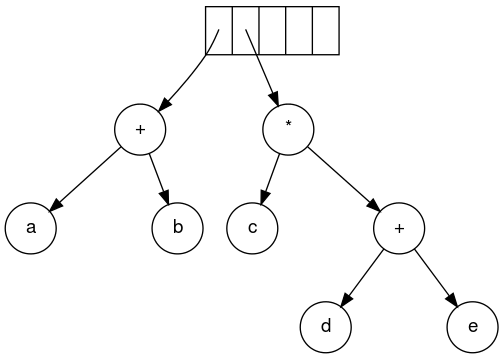
Next, c, d, and e are read. A one-node tree is created for each and a pointer to the corresponding tree is pushed onto the stack.



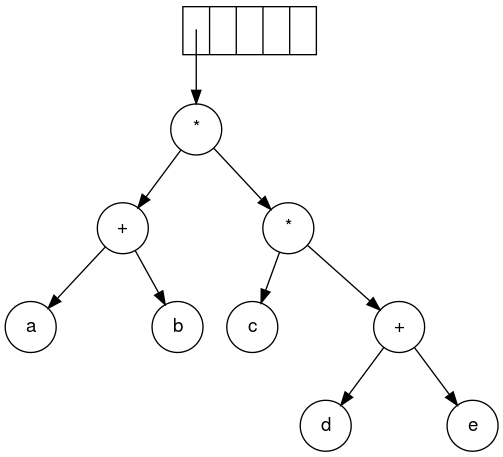
Continuing, a '+' is read, and it merges the last two trees.



Now, a '\*' is read. The last two tree pointers are popped and a new tree is formed with a '\*' as the root.



Finally, the last symbol is read. The two trees are merged and a pointer to the final tree remains on the stack.



**References**

[**file:///C:/hjt/BST\_Deletion.pdf**](file:///C:/hjt/BST_Deletion.pdf)

<http://courses.cs.vt.edu/cs2604/spring02/Notes/C08.GeneralTrees.pdf>

http://btechsmartclass.com/DS/U3\_T1.html

<http://www.cs.kau.se/cs/education/courses/dvgb03/lectures/NTrees_1.pdf>

<http://thecodegallery.com/DSM/GeneralTree.php>

<http://www.ggu.ac.in/download/Class-Note13/ds%20lecture%20notes3-25.10.13.pdf>

<https://www.tutorialspoint.com/data_structures_algorithms/binary_search_tree.htm>

http://see-programming.blogspot.in/2013/05/insertion-deletion-traversal-in.html

**video links**

<https://www.youtube.com/watch?v=9RHO6jU--GU> (for tree traversal-BFS)