CANARA ENGINEERING COLLEGE

Benjanapadavu, Bantwal Taluk - 574219

Department of Computer Science & Engineering

VISION

To be recognized as a center of knowledge dissemination in Computer Science and Engineering by imparting value-added education to transform budding minds into competent computer professionals.

MISSION

M1. Provide a learning environment enriched with ethics that helps in enhancing problem solving skills of students and, cater to the needs of the society and industry.

M2. Expose the students to cutting-edge technologies and state-of-the-art tools in the many areas of Computer Science & Engineering.

M3. Create opportunities for all round development of students through co-curricular and extra-curricular activities.

M4. Promote research, innovation and development activities among staff and students.

PROGRAMME EDUCATIONAL OBJECTIVES

PE01: Graduates will work productively as computer science engineers exhibiting ethical qualities and leadership roles in multidisciplinary teams.

PEO2: Graduates will adapt to the changing technologies, tools and societal requirements.

PEO3: Graduates will design and deploy software that meets the needs of individuals and the industries

PEO4: Graduates will take up higher education and/or be associated with the field so that they can keep themselves abreast of Research & Development

PROGRAMME OUTCOMES

Engineering graduates in Computer Science and Engineering will be able to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specific needs with appropriate consideration for the public health and safety, and the cultural, societal and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods, including design of experiments, analysis and interpretation of data and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Select/Create and apply appropriate techniques, resources and modern engineering and IT tools, including prediction and modeling to complex engineering activities, taking comprehensive cognizance of their limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and Sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts and demonstrate the knowledge of and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the relevant scientific and/or engineering practices.
9. **Individual and team work:** Function effectively as an individual and as a member or leader in diverse teams and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with the society-at-large, such as being able to comprehend and write effective reports and design documentation, make effective presentations and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work as a member and leader in a team to manage projects and in multidisciplinary environments.

12. **Life-long learning:** Recognize the need for and above have the preparation and ability to engage in independent and life-long learning in the broadcast context of technological changes.

PROGRAMME SPECIFIC OUTCOMES

1. **Computer System Components:** Apply the principles of computer system architecture and software to design, develop and deploy computer subsystem.
2. **Data Driven and Internet Applications:** Apply the knowledge of data storage, analytics and network architecture in designing Internet based applications.

<table>
<tr><th colspan="4">DATA STRUCTURES AND APPLICATIONS
(Effective from the academic year 2018 -2019)
SEMESTER – III</th></tr>
<tr><td>Course Code</td><td>18CS32</td><td>CIE Marks</td><td>40</td></tr>
<tr><td>Number of Contact Hours/Week</td><td>3:2:0</td><td>SEE Marks</td><td>60</td></tr>
<tr><td>Total Number of Contact Hours</td><td>50</td><td>Exam Hours</td><td>03</td></tr>
<tr><td colspan="4" align="center">CREDITS –4</td></tr>
<tr><td colspan="4">Course Learning Objectives: This course (18CS32) will enable students to:</td></tr>
<tr><td colspan="4">Explain fundamentals of data structures and their applications essential for programming/problem solving.
Illustrate linear representation of data structures: Stack, Queues, Lists, Trees and Graphs.
Demonstrate sorting and searching algorithms.
Find suitable data structure during application development/Problem Solving.</td></tr>
</table>

<table>
<tr><th>Module 1</th><th>Contact Hours</th></tr>
<tr><td>Introduction: Data Structures, Classifications (Primitive & Non-Primitive), Data structure Operations, Review of Arrays, Structures, Self-Referential Structures, and Unions. Pointers and Dynamic Memory Allocation Functions. Representation of Linear Arrays in Memory, Dynamically allocated arrays.
Array Operations: Traversing, inserting, deleting, searching, and sorting. Multidimensional Arrays, Polynomials and Sparse Matrices.
Strings: Basic Terminology, Storing, Operations and Pattern Matching algorithms. Programming Examples.
Textbook 1: Chapter 1: 1.2, Chapter 2: 2.2 - 2.7 Text Textbook 2: Chapter 1: 1.1 - 1.4,
Chapter 3: 3.1 - 3.3, 3.5, 3.7, Chapter 4: 4.1 - 4.9, 4.14 Reference 3: Chapter 1: 1.4 RBT: L1, L2, L3</td><td>10</td></tr>
</table>

Module 2	
Stacks: Definition, Stack Operations, Array Representation of Stacks, Stacks using Dynamic Arrays, Stack Applications: Polish notation, Infix to postfix conversion, evaluation of postfix expression. **Recursion** - Factorial, GCD, Fibonacci Sequence, Tower of Hanoi, Ackerman's function. **Queues:** Definition, Array Representation, Queue Operations, Circular Queues, Circular queues using Dynamic arrays, Dequeues, Priority Queues, A Mazing Problem. Multiple Stacks and Queues. Programming Examples. **Textbook 1: Chapter 3: 3.1 -3.7 Textbook 2: Chapter 6: 6.1 -6.3, 6.5, 6.7-6.10, 6.12, 6.13** **RBT: L1, L2, L3**	10
Module 3	
Linked Lists: Definition, Representation of linked lists in Memory, Memory allocation; Garbage Collection. Linked list operations: Traversing, Searching, Insertion, and Deletion. Doubly Linked lists, Circular linked lists, and header linked lists. Linked Stacks and Queues. Applications of Linked lists – Polynomials, Sparse matrix representation. Programming Examples **Textbook 1: Chapter 4: 4.1 – 4.6, 4.8, Textbook 2: Chapter 5: 5.1 – 5.10,** **RBT: L1, L2, L3**	10
Module 4	
Trees: Terminology, Binary Trees, Properties of Binary trees, Array and linked Representation of Binary Trees, Binary Tree Traversals - Inorder, postorder, preorder; Additional Binary tree operations. Threaded binary trees, Binary Search Trees – Definition, Insertion, Deletion, Traversal, Searching, Application of Trees-Evaluation of Expression, Programming Examples **Textbook 1: Chapter 5: 5.1 –5.5, 5.7; Textbook 2: Chapter 7: 7.1 – 7.9** **RBT: L1, L2, L3**	10
Module 5	
Graphs: Definitions, Terminologies, Matrix and Adjacency List Representation of Graphs, Elementary Graph operations, Traversal methods: Breadth First Search and Depth First Search. **Sorting and Searching**: Insertion Sort, Radix sort, Address Calculation Sort. **Hashing:** Hash Table organizations, Hashing Functions, Static and Dynamic Hashing. **Files and Their Organization:** Data Hierarchy, File Attributes, Text Files and Binary Files, Basic File Operations, File Organizations and Indexing **Textbook 1: Chapter 6: 6.1 –6.2, Chapter 7:7.2, Chapter 8: 8.1-8.3** **Textbook 2: Chapter 8: 8.1 – 8.7, Chapter 9: 9.1-9.3, 9.7, 9.9** **Reference 2: Chapter 16: 16.1 - 16.7** **RBT: L1, L2, L3**	10
Course Outcomes: The student will be able to:	

- Use different types of data structures, operations and algorithms
- Apply searching and sorting operations on files
- Use stack, Queue, Lists, Trees and Graphs in problem solving
- Implement all data structures in a high-level language for problem solving.

Question Paper Pattern:

- The question paper will have ten questions.
- Each full Question consisting of 20 marks
- There will be 2 full questions (with a maximum of four sub questions) from each module.
- Each full question will have sub questions covering all the topics under a module.
- The students will have to answer 5 full questions, selecting one full question from each module.

Textbooks:

1. Ellis Horowitz and Sartaj Sahni, Fundamentals of Data Structures in C, 2nd Ed, Universities Press, 2014.
2. Seymour Lipschutz, Data Structures Schaum's Outlines, Revised 1st Ed, McGraw Hill, 2014.

Reference Books:

1. Gilberg & Forouzan, Data Structures: A Pseudo-code approach with C, 2nd Ed, Cengage Learning,2014.
2. Reema Thareja, Data Structures using C, 3rd Ed, Oxford press, 2012.
3. Jean-Paul Tremblay & Paul G. Sorenson, An Introduction to Data Structures with Applications, 2nd Ed, McGraw Hill, 2013
4. A M Tenenbaum, Data Structures using C, PHI, 1989
5. Robert Kruse, Data Structures and Program Design in C, 2nd Ed, PHI, 1996.

COURSE OBJECTIVES:

This course will enable students to	
1	Explain fundamentals of data structures and their applications essential for programming/problem solving.
2	Analyze Linear Data Structures: Stack, Queues, Lists
3	Analyze Non-Linear Data Structures: Trees, Graphs
4	Analyze and Evaluate the sorting & searching algorithms
5	Assess appropriate data structure during program development/Problem Solving

COURSE OUTCOMES (COs):

SL. NO	DESCRIPTION
	The students are able to:
CO:1	Explain various types of data structures, sorting and searching operations on arrays.
CO:2	Develop the programs on operations like searching, insertion, deletion, traversing mechanism on stack and queues.
CO:3	Apply the basic knowledge of linked list to solve real world problems.
CO:4	Develop the programs on operations like searching, insertion, deletion, traversing mechanism on trees.
CO:5	Explain the basic graph algorithms and their analyses. Employ graphs and Sorting and searching operations, to model engineering problems, when appropriate

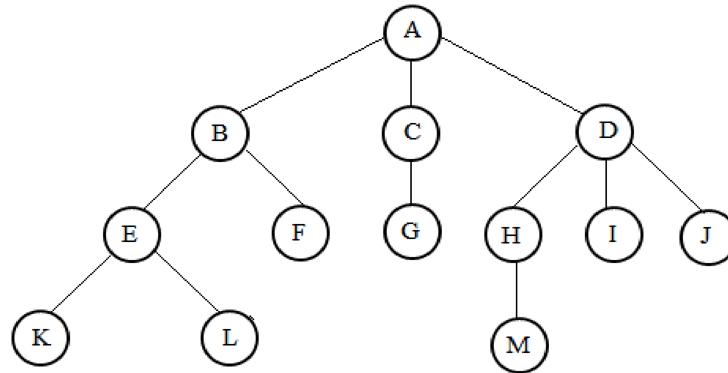
Contents

Sl. No.	Topic	Page No.
1	Trees: terminology	2
2	Representation of Trees	3
3	Binary trees	6
4	Properties of binary trees	8
5	Binary tree representation	9
6	Binary tree traversals	12
7	Additional binary tree operations	16
8	Threaded binary tree	19
9	Application of Trees-Evaluation of Expression	27
10	Web links	28
11	Question bank	29
12	References	31

TREES

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

- There is a specially designated node called *root*.
- The remaining nodes are partitioned into $n >= 0$ disjoint set $T_1,\ldots,T_n$, where each of these sets is a tree. $T_1,\ldots,T_n$ are called the *subtrees* of the root.

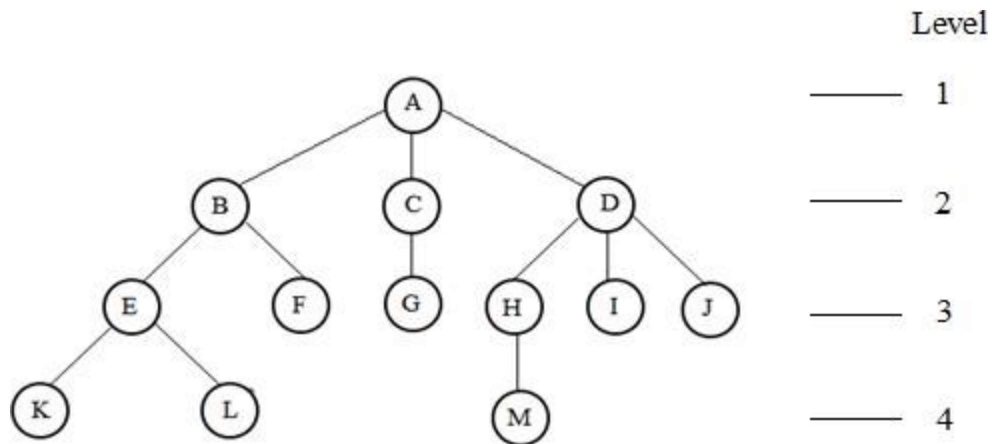


Every node in the tree is the root of some subtree

TERMINOLOGY

- ***Node:*** The item of information plus the branches to other nodes
- ***Degree:*** The number of subtrees of a node
- ***Degree of a tree:*** The maximum of the degree of the nodes in the tree.
- ***Terminal nodes (or leaf):*** nodes that have degree zero or node with no successor
- ***Nonterminal nodes***: nodes that don't belong to terminal nodes.
- ***Parent and Children:*** Suppose N is a node in T with left successor S_1 and right successor S_2, then N is called the Parent of S_1 and S_2. Here, S_1 is called left child and S_2 is called right child of N.
- ***Siblings:*** Children of the same parent are said to be siblings.
- ***Edge:*** A line drawn from node N of a T to a successor is called an edge.
- ***Path:*** A sequence of consecutive edges from node N to a node M is called a path.
- ***Ancestors of a node:*** All the nodes along the path from the root to that node.
- ***The level of a node:*** defined by letting the root be at level one. If a node is at level l, then it children are at level $l+1$.
- ***Height (or depth):*** The maximum level of any node in the tree

Example



A is the root node

B is the parent of E and F

C and D are the sibling of

B

E and F are the children of

B

K, L, F, G, M, I, J are external nodes, or leaves

A, B, C, D, E, H are internal nodes

The level of E is 3

The height (depth) of the tree is

4 The degree of node B is 2

The degree of the tree is 3

The ancestors of node M is A, D, H

The descendants of node D is H, I, J, M

Representation of Trees

There are several ways to represent a given tree such as:

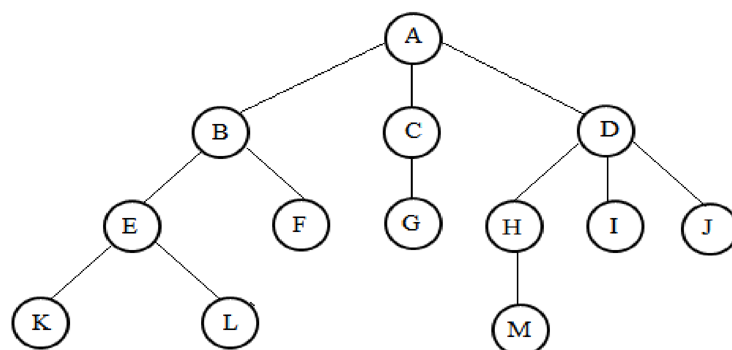


Figure (A)

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

List Representation:

The tree can be represented as a List. The tree of **figure (A)** could be written as the list.

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

- The information in the root node comes first.
- The root node is followed by a list of the subtrees of that node.

Tree node is represented by a memory node that has fields for the data and pointers to the tree node's children

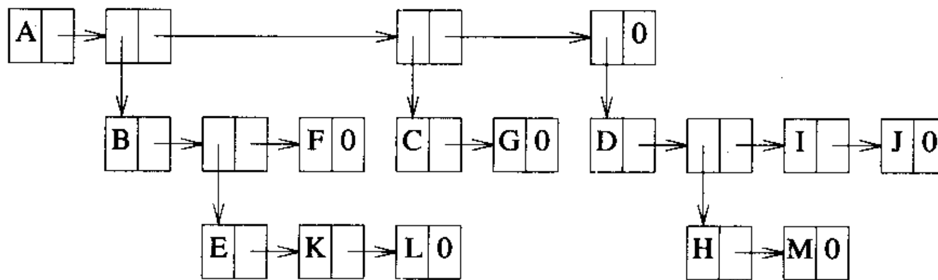


Figure (B): List representation of the tree of figure (A)

- Since the degree of each tree node may be different, so memory nodes with a varying number of pointer fields are used.
- For a tree of degree k, the node structure can be represented as below figure. Each child field is used to point to a subtree.

Left Child-Right Sibling Representation

The below figure show the node structure used in the left child-right sibling representation

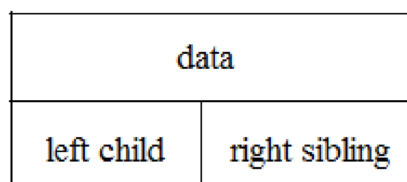


Figure (c): Left child right sibling node structure

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

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

Ex:

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

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

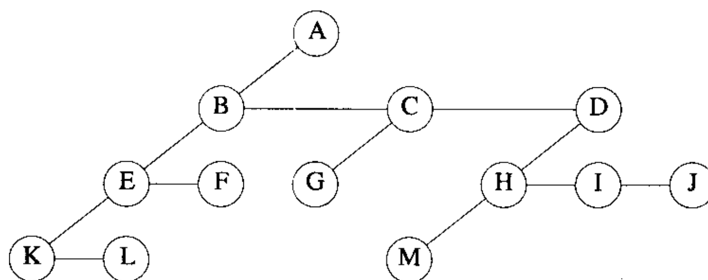


Figure (D): Left child-right sibling representation of tree of figure (A)

Representation as a Degree-Two Tree

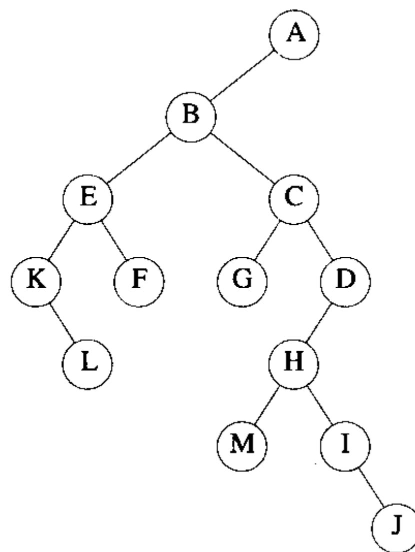


Figure (E): degree-two representation

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

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

BINARY TREES

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

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

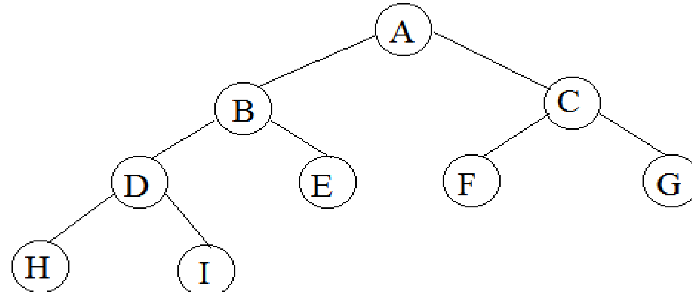


Figure: Binary Tree

Different kinds of Binary Tree

1. Skewed Tree

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

or

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

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

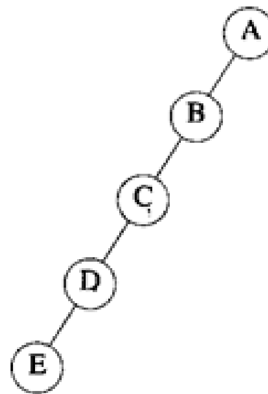


Figure (a): Skewed binary tree

2. Complete Binary Tree

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

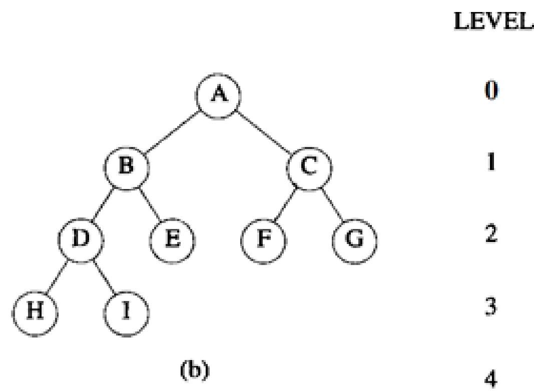


Figure (b): Complete binary tree

3. Full Binary Tree

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

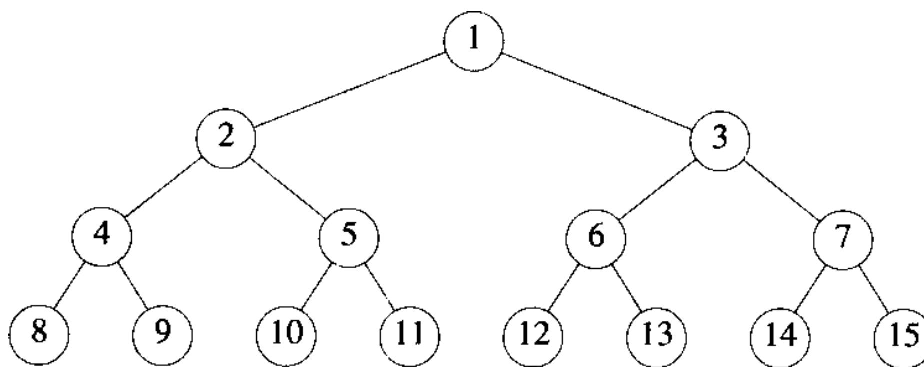
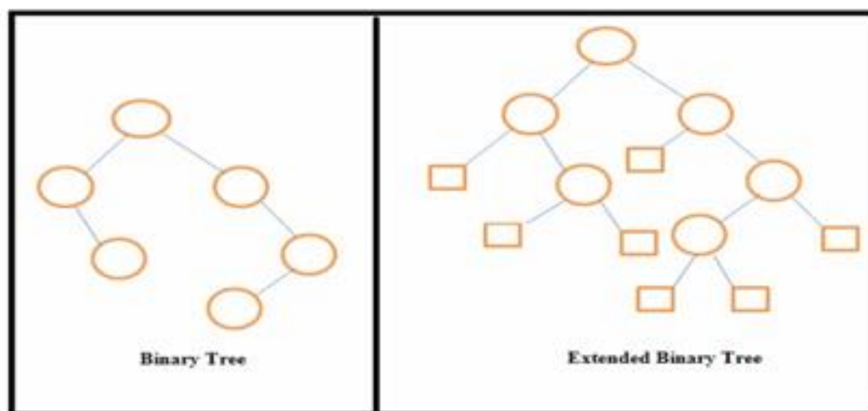


Figure: Full binary tree of level 4 with sequential node number

4. Extended Binary Trees or 2-trees

An *extended binary tree* is a transformation of any binary tree into a complete binary tree. This transformation consists of replacing every null subtree of the original tree with “special nodes.” The nodes from the original tree are then *internal nodes*, while the special nodes are *external nodes*.

For instance, consider the following binary tree.



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

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

PROPERTIES OF BINARY TREES

Lemma 1: [Maximum number of nodes]:

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

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

Proof:

(1) The proof is by induction on i.

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

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

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

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

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

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

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

Proof: Let ***n₁*** be the number of nodes of degree one and ***n*** the total number of nodes.

Since all nodes in T are at most of degree two, we have

$$n = n_0 + n_1 + n_2 \qquad (1)$$

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

$$n = B + 1.$$

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

$$B = n_1 + 2n_2.$$

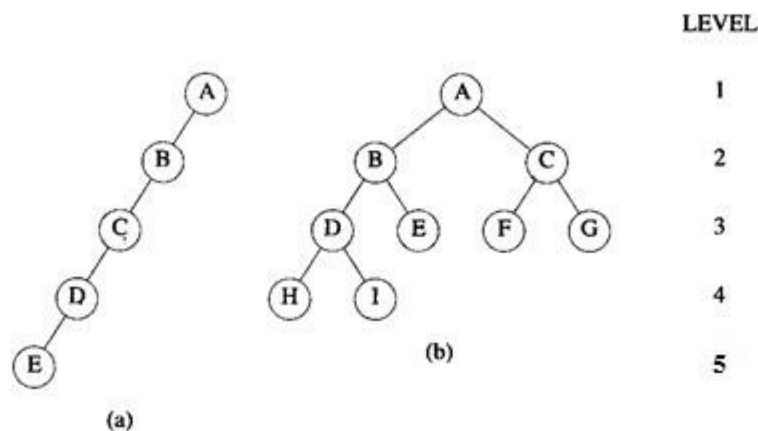
Hence, we obtain

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

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

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

Consider the figure:



Here, For Figure (b) n_2=4, n_0= n_2+1= 4+1=5

Therefore, the total number of leaf node=5

BINARY TREE REPRESENTATION

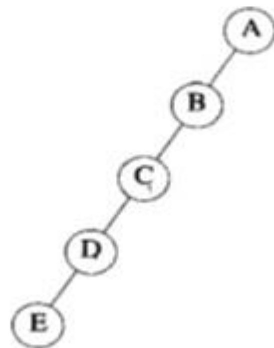
The storage representation of binary trees can be classified as

1. Array representation
2. Linked representation.

Array representation:

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

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



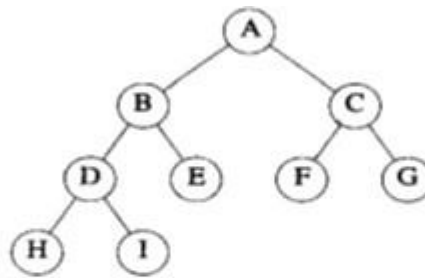


Figure 1(b) Complete binary tree

Figure 1(a) Skewed binary tree

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

(a). Tree of figure 1(a)

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

(b). Tree of figure 1(b)

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

Linked representation:

The problems in array representation are:

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

These problems can be easily overcome by linked representation.

Each node has three fields,

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

C Code for node:

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

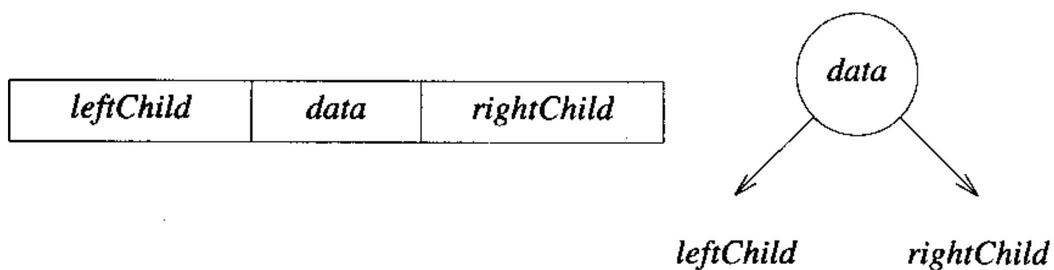


Figure: Node representation

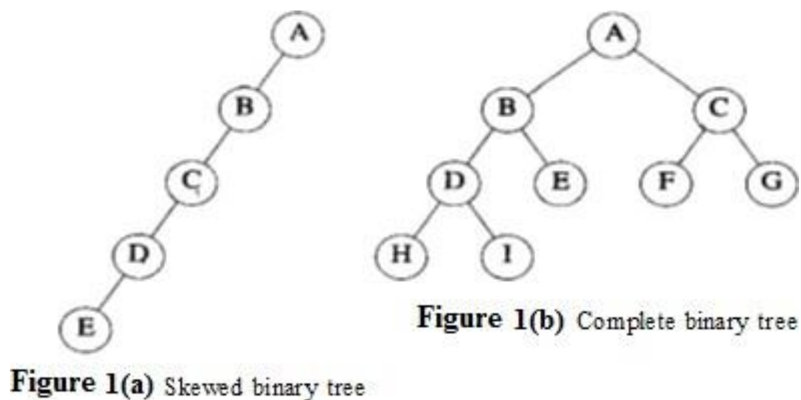
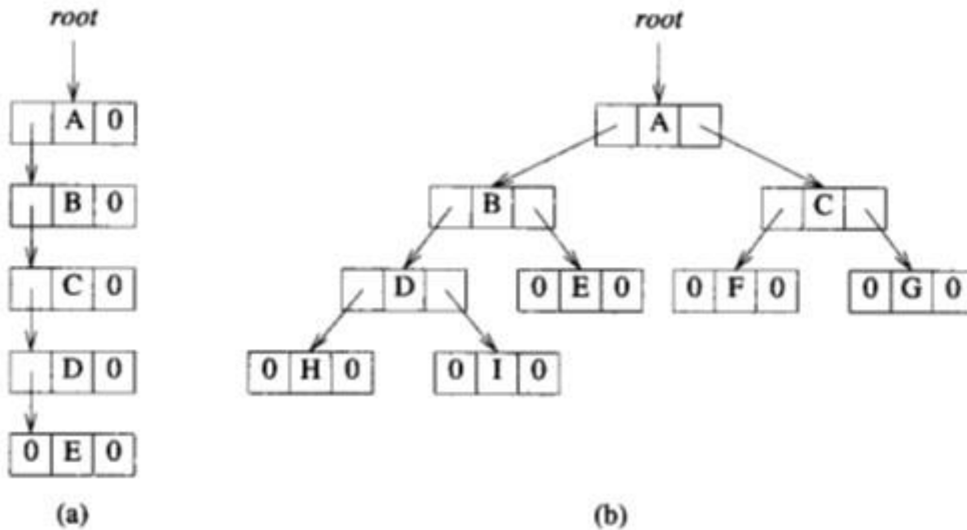


Figure 1(a) Skewed binary tree

Figure 1(b) Complete binary tree



Linked representation of the binary tree

BINARY TREE TRAVERSALS

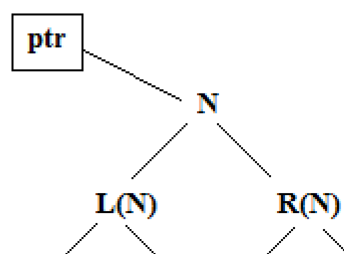
Visiting each node in a tree exactly once is called tree traversal

The different methods of traversing a binary tree are:

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

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

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



Recursion function:

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

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

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

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

Recursion function:

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

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

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

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

Recursion function:

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

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

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

4. Iterative inorder Traversal:

Iterative inorder traversal explicitly make use of stack function.

The left nodes are pushed into stack until a null node is reached, the node is then removed from the stack and displayed, and the node’s right child is stacked until a null node is reached. The traversal then continues with the left child. The traversal is complete when the stack is empty.

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

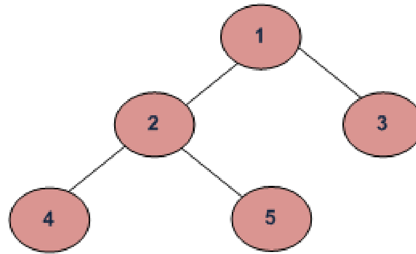
Program : Iterative inorder traversal

5. Level-Order traversal:

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

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

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



Level order traversal: 1 2 3 4 5

Initially in the code for level order add the root to the queue. The function operates by deleting the node at the front of the queue, printing the nodes data field and adding the nodes left and right children to the queue.

Function for level order traversal of a binary tree:

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

Program : Level-order traversal of a binary tree

6. Traversal without a stack

- **Add a parent field to each node.**

ADDITIONAL BINARY TREE OPERATIONS

1. Copying a Binary tree

This operations will perform a copying of one binary tree to another.

C function to copy a binary tree:

```
treepointer copy(treepointer original)
{
    if (original)
    {
        malloc(temp,sizeof(*temp));
        temp→leftchild=copy(original→leftchild);
        temp→rightchild=copy(original→rightchild)
        ; temp→data=original→data;
        return temp;
    }
    return NULL;
}
```

2. Testing Equality

This operation will determine the equivalence of two binary tree. Equivalence binary tree have the same structure and the same information in the corresponding nodes.

C function for testing equality of a binary tree: int equal(treepointer first, treepointer second)

```
{
        return((!first && !second) || (first && second && (first→data==second→data) &&
        equal(first→leftchild,second→leftchild) && equal(first→rightchild, second→rightchild))
}
```

This function will return TRUE if two trees are equivalent and FALSE if they are not.

3. The Satisfiability problem

- Consider the formula that is constructed by set of variables: x_1, x_2, …, x_n and operators ∧ (*and*), ∨ (*or*), ¬ (*not*).
- The variables can hold only of two possible values, *true* or *false*.
- The expression can form using these variables and operators is defined by the following rules.
    - A variable is an expression
    - If x and y are expressions, then $\neg x$, $x \wedge y$, $x \vee y$ are expressions
    - Parentheses can be used to alter the normal order of evaluation (¬ > ∧ > ∨)

Example: $x1 \vee (x2 \wedge \neg x3)$ If $x1$ and $x3$ are *false* and $x2$ is *true*

$= false \vee (true \wedge \neg false)$

$= false \vee true$

$= true$

The satisfiability problem for formulas of the propositional calculus asks if there is an assignment of values to the variable that causes the value of the expression to be *true*.

Let's assume the formula in a binary tree

$$(x1 \wedge \neg x2) \vee (\neg x1 \wedge x3) \vee \neg x3$$

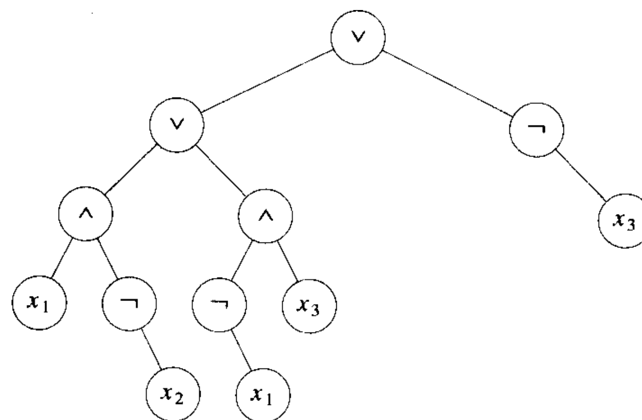


Figure : Propositional formula in a binary tree

The inorder traversal of this tree is

$$x1 \wedge \neg x2 \vee \neg x1 \wedge x3 \vee \neg x3$$

The algorithm to determine satisfiability is to let (x1, x2, x3) takes on all the possible combination of true and false values to check the formula for each combination.

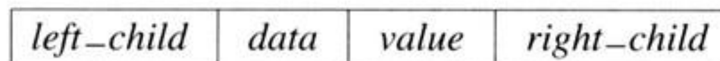
For *n* value of an expression, there are 2^n possible combinations of *true* and *false*

For example n=3, the eight combinations are (t,t,t), (t,t,f), (t,f,t), (t,f,f), (f,t,t), (f,t,f), (f,f,t), (f,f,f).

The algorithm will take O(g 2^n), where g is the time to substitute values for $x_1, x_2, \ldots x_n$ and evaluate the expression.

Node structure:

For the purpose of evaluation algorithm, assume each node has four fields:



left_child	*data*	*value*	*right_child*

Figure : Node structure for the satisfiability problem

Define this node structure in C as:

```
typedef enum {not,and,or,true,false} logical;
typedef struct node *tree_pointer;
typedef struct node {
        tree_pointer left_child;
        logical      data;
        short int    value;
        tree_pointer right_child;
        } ;
```

Satisfiability function: The first version of Satisfiability algorithm

```
for (all 2^n possible combinations) {
   generate the next combination;
   replace the variables by their values;
   evaluate root by traversing it in postorder;
   if (root->value) {
      printf(<combination>);
      return;
   }
}
printf("No satisfiable combination\n");
```

THREADED BINARY TREE

The limitations of binary tree are:

- In binary tree, there are n+1 null links out of 2n total links.
- Traversing a tree with binary tree is time consuming.

These limitations can be overcome by threaded binary tree.

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

To construct the threads use the following rules:

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

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

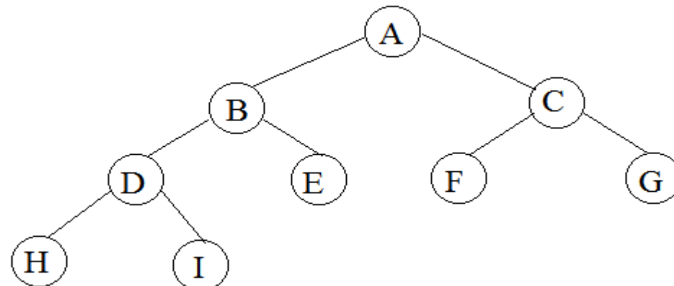


Figure A: Binary Tree

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

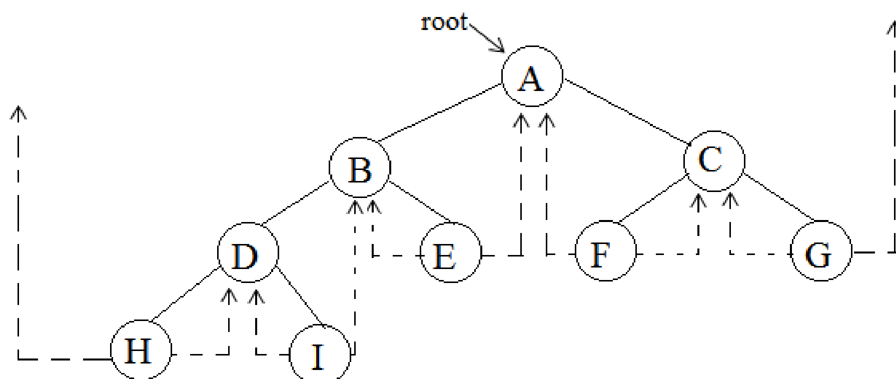


Figure B: Threaded tree corresponding to Figure A

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

When trees are represented in memory, it should be able to distinguish between threads and pointers. This can be done by adding two additional fields to node structure, ie., *leftThread* and *rightThread*

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

Node Structure:

The node structure is given in C declaration

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

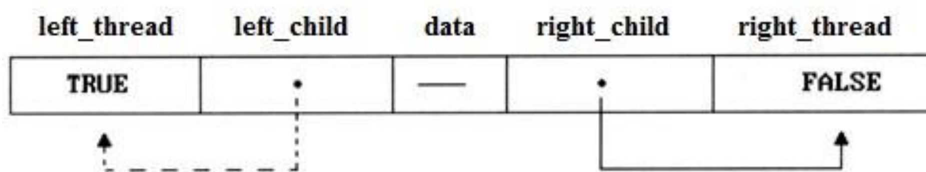
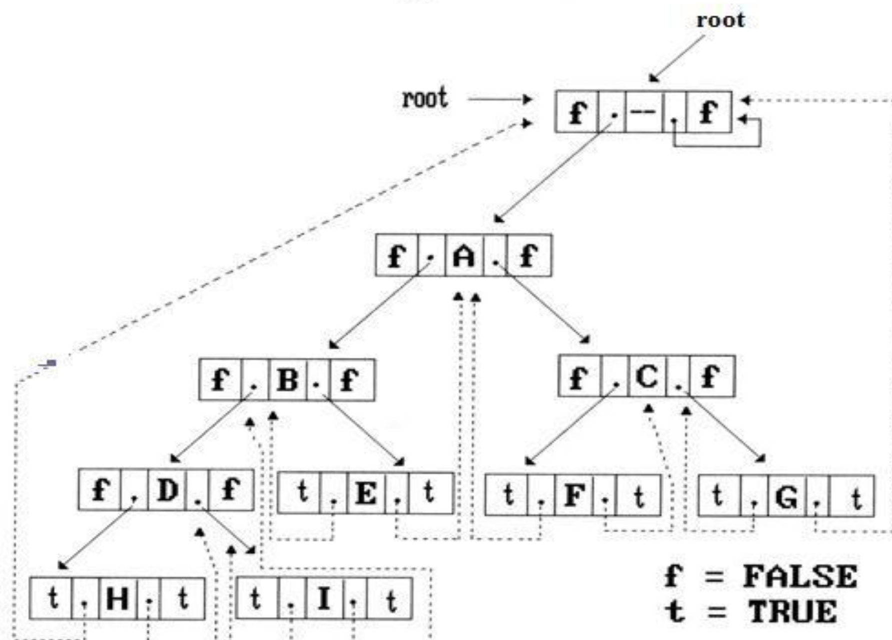


Figure An empty threaded tree



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

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

<u>Inorder Traversal of a Threaded Binary Tree</u>

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

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

Program: Finding inorder successor of a node

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

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

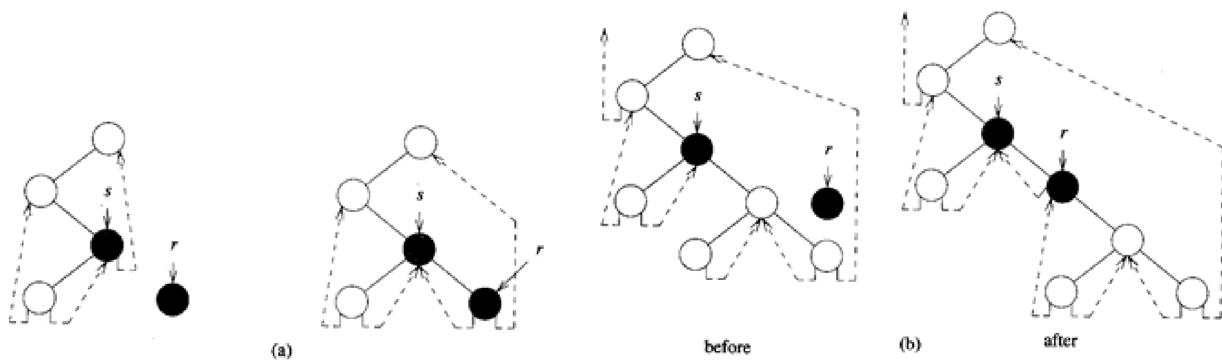
Program: Inorder traversal of a threaded binary tree

Inserting a Node into a Threaded Binary Tree

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

The cases for insertion are:

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




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

Binary search trees (**BST**), sometimes called **ordered** or **sorted binary trees**, are a particular type of containers: data structures that store "items" (such as numbers, names etc.) in memory. They allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its *key* (e.g., finding the phone number of a person by name).

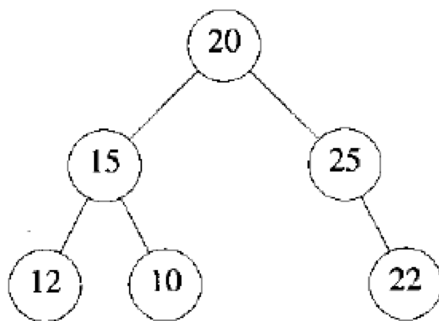
Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations on hash tables.

(1) Each node has exactly one key and the keys in the tree are distinct.

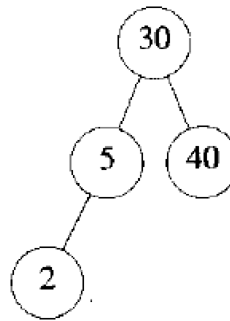
(2) The keys (if any) in the left subtree are smaller than the key in the root.

(3) The keys (if any) in the right subtree are larger than the key in the root.

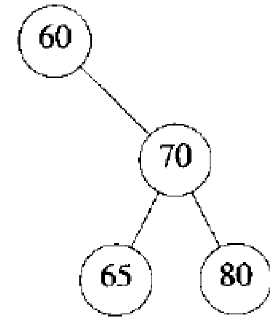
(4) The left and right subtrees are also binary search trees. □



(a)



(b)



(c)

Operations

Binary search trees support three main operations: insertion of elements, deletion of elements, and lookup (checking whether a key is present).

Searching

Searching a binary search tree for a specific key can be a programmed recursively or iteratively.

We begin by examining the root node. If the tree is *null*, the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and we return the node.

If the key is less than that of the root, we search the left subtree. Similarly, if the key is greater than that of the root, we search the right subtree. This process is repeated until the key is found or the remaining subtree is *null*. If the searched key is not found before a *null* subtree is reached, then the key is not present in the tree.

```
element* search(treePointer root, int key)
{/* return a pointer to the element whose key is k, if
    there is no such element, return NULL. */
  if (!root) return NULL;
  if (k == root→data.key) return &(root→data);
  if (k < root→data.key)
    return search(root→leftChild, k);
  return search(root→rightChild, k);
}
```

The same algorithm can be implemented iteratively:

```
element* iterSearch(treePointer tree, int k)
{/* return a pointer to the element whose key is k,  if
    there is no such element, return NULL. */
  while (tree) {
    if (k == tree→data.key) return &(tree→data);
    if (k < tree→data.key)
      tree = tree→leftChild;
    else
      tree = tree→rightChild;
    }
  return NULL;
}
```

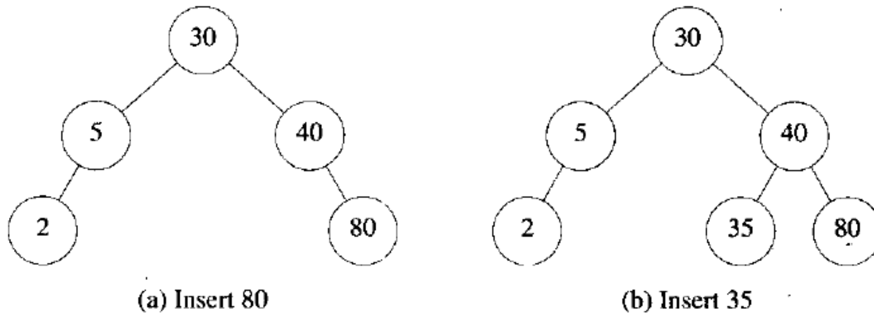
These two examples rely on the order relation being a total order.

If the order relation is only a total preorder a reasonable extension of the functionality is the following: also in case of equality search down to the leaves in a direction specifiable by the user. A binary tree sort equipped with such a comparison function becomes stable.

Because in the worst case this algorithm must search from the root of the tree to the leaf farthest from the root, the search operation takes time proportional to the tree's *height* (see tree terminology). On average, binary search trees with n nodes have O(log n) height. However, in the worst case, binary search trees can have O(n) height, when the unbalanced tree resembles a linked list (degenerate tree).

Insertion

Insertion begins as a search would begin; if the key is not equal to that of the root, we search the left or right subtrees as before. Eventually, we will reach an external node and add the new key-value pair (here encoded as a record 'newNode') as its right or left child, depending on the node's key. In other words, we examine the root and recursively insert the new node to the left subtree if its key is less than that of the root, or the right subtree if its key is greater than or equal to the root.



(a) Insert 80 (b) Insert 35

Here's how a typical binary search tree insertion might be performed in a binary tree in C++:

```
void insert(treePointer *node, int k, iType theItem)
{/* if k is in the tree pointed at by node do nothing;
    otherwise add a new node with data = (k, theItem) */
   treePointer ptr, temp = modifiedSearch(*node, k);
   if (temp || !(*node)) {
      /* k is not in the tree */
      MALLOC(ptr, sizeof(*ptr));
      ptr→data.key = k;
      ptr→data.item = theItem;
      ptr→leftChild = ptr→rightChild = NULL;
      if (*node) /* insert as child of temp */
         if (k < temp→data.key) temp→leftChild = ptr;
         else temp→rightChild = ptr;
      else *node = ptr;
   }
}
```

The part that is rebuilt uses O(log *n*) space in the average case and O(*n*) in the worst case.

In either version, this operation requires time proportional to the height of the tree in the worst case, which is O(log *n*) time in the average case over all trees, but O(*n*)time in the worst case.

Another way to explain insertion is that in order to insert a new node in the tree, its key is first compared with that of the root. If its key is less than the root's, it is then compared with the key of the root's left child. If its key is greater, it is compared with the root's right child. This process continues, until the new node is compared with

a leaf node, and then it is added as this node's right or left child, depending on its key: if the key is less than the leaf's key, then it is inserted as the leaf's left child, otherwise as the leaf's right child.

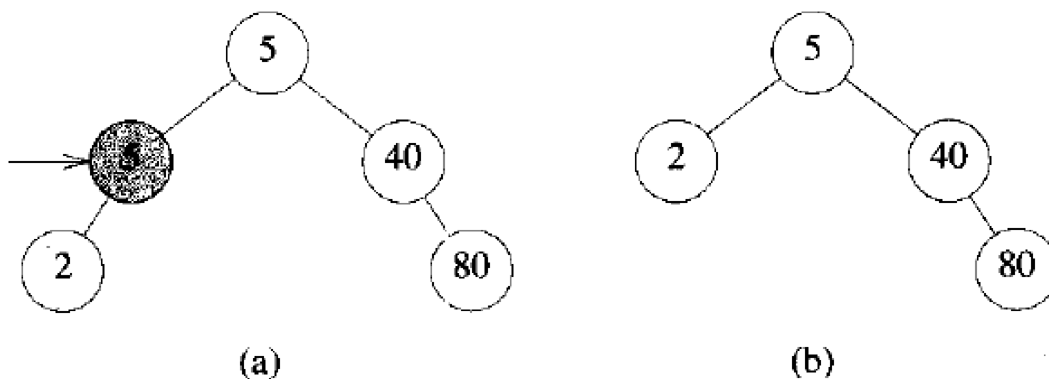
There are other ways of inserting nodes into a binary tree, but this is the only way of inserting nodes at the leaves and at the same time preserving the BST structure.

Deletion

There are three possible cases to consider:

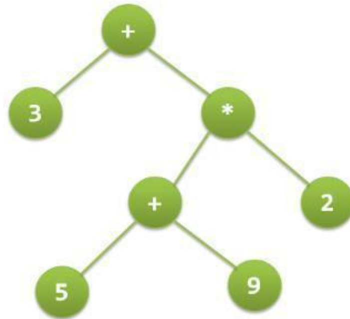
- Deleting a node with no children: simply remove the node from the tree.
- Deleting a node with one child: remove the node and replace it with its child.
- Deleting a node with two children: call the node to be deleted *N*. Do not delete *N*. Instead, choose either its in-order successor node or its in-order predecessor node, *R*. Copy the value of *R* to *N*, then recursively call delete on *R* until reaching one of the first two cases. If you choose in-order successor of a node, as right sub tree is not NIL (Our present case is node has 2 children), then its in-order successor is node with least value in its right sub tree, which will have at a maximum of 1 sub tree, so deleting it would fall in one of the first 2 cases.

Broadly speaking, nodes with children are harder to delete. As with all binary trees, a node's in-order successor is its right subtree's left-most child, and a node's in-order predecessor is the left subtree's right-most child. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.



Expression tree

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand so for example expression tree for 3 + ((5+9)*2) would be:



Inorder traversal of expression tree produces infix version of given postfix expression (same with preorder traversal it gives prefix expression)

Evaluating the expression represented by expression tree:

Let t be the expression tree

If t is not null then

If t.value is operand then

Return t.value

A = solve(t.left)

B = solve(t.right)

// calculate applies operator 't.value'

// on A and B, and

returns value

Return calculate(A,

B, t.value)

Construction of Expression Tree:

Now For constructing expression tree we use a stack. We loop through input expression and do following for every character.

1) If character is operand push that into stack

2) If character is operator pop two values from stack make them its child and push current node again. At the end only element of stack will be root of expression tree.

Web Links

[1] Trees: Basics of Trees, Terminology (M4 L1)
[https://www.youtube.com/watch?v=pza1sNBPcUU&list=PLVDfFatHsysQGtvuaDbTTkle69C0wHaK_&index=28]

[2] Trees: Representation of Trees (M4 L2)
[https://www.youtube.com/watch?v=ScTbMqQVIgI&list=PLVDfFatHsysQGtvuaDbTTkle69C0wHaK_&index=29]

[3] Trees: Binary Trees, Different kinds of Binary Tree, Properties of Binary Trees(M4 L3)
[https://www.youtube.com/watch?v=wUZuSdZQ-f8&list=PLVDfFatHsysQGtvuaDbTTkle69C0wHaK_&index=30]

[4] Trees: BINARY TREE REPRESENTATION (M4L4)
[https://www.youtube.com/watch?v=4SFGVBhe-dY&list=PLVDfFatHsysQGtvuaDbTTkle69C0wHaK_&index=31]

[5] Trees: BINARY TREE TRAVERSAL (M4 L5)
[https://www.youtube.com/watch?v=eeBJ83cgZwk&list=PLVDfFatHsysQGtvuaDbTTkle69C0wHaK_&index=32]

[6] Trees: Binary Tree Construction Given:Inorder&Postorder,Inorder&PreorderTravesal(M4L6)
[https://www.youtube.com/watch?v=Y4jaSrm_pkA&list=PLVDfFatHsysQGtvuaDbTTkle69C0wHaK_&index=33]

[7] Trees : Additional Binary Tree Operations (M4 L7)
[https://www.youtube.com/watch?v=hCUrgRTmb5w&list=PLVDfFatHsysQGtvuaDbTTkle69C0wHaK_&index=34]

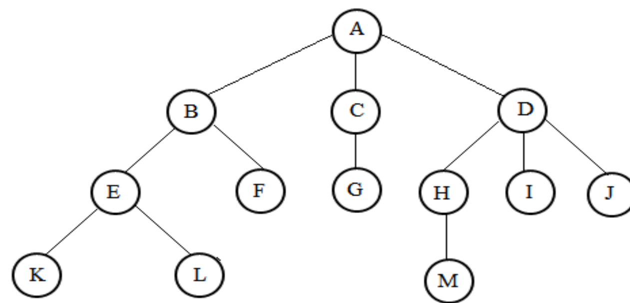
[8] Trees: THREADED BINARY TREE (M4 L8)
[https://www.youtube.com/watch?v=hvcDjNODW0I&list=PLVDfFatHsysQGtvuaDbTTkle69C0wHaK_&index=35]

[9] Trees: Binary Search Trees–Definition, Insertion, Deletion, Traversal, Searching(M4L9)
[https://www.youtube.com/watch?v=g8o4w5Tn7Gs&list=PLVDfFatHsysQGtvuaDbTTkle69C0wHaK_&index=36]

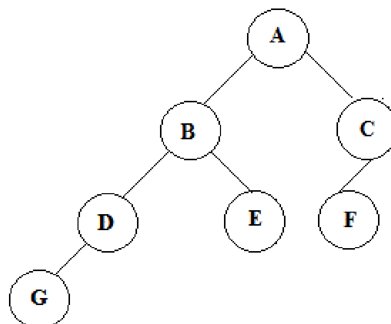
[10] Trees: Application of Trees-Evaluation of Expression (M4L10)
[https://www.youtube.com/watch?v=qkVAtli8yRc&list=PLVDfFatHsysQGtvuaDbTTkle69C0wHaK_&index=37]

Question Bank

1. Define tree? For a given tree explain the following
   i) Root node
   ii) Degree
   iii) Sibling
   iv) Depth of a tree
   v) Ancestor



2. With an example, explain the different types of representation of tree.
3. Define a binary tree. With example show array and linked representation of binary tree.
4. Write an expression tree for an expression i) A / B + C * D + E ii) ((6+(3-2)*5)^2+3)
5. Mention different types of binary trees and explain briefly
6. State the properties of a binary tree in detail
7. Write the C-routines to traverse the given tree using
   i) Inorder
   ii) Pre-order
   iii) Post-order



8. Construct the binary tree (B-tree) from the given traversals:

	i)	ii)	iii)
Preorder:	ABDCEF	/ + * 1 $ 2 3 4 5	A B D G C E H I F
In-order:	BDAEFC	1 + 2 * 3 $ 4 – 5	D G B A H E I C F
Postorder:	DBFECA		

9. Explain iterative inorder traversal and level order traversal with C routines.
10. List and discuss the additional operations of binary tree with appropriate examples.
11. What is threaded binary tree? Write the rules to construct the threads.
12. Write the node structures and C declaration of threaded binary tree.
13. Explain inorder traversal and insertion of a node in a threaded binary tree.
14. Describe the binary search tree with an example. Write the iterative and recursive function to search for a key value in a binary search tree
15. Construct a binary search tree for the inputs i) 22, 14, 18, 50, 9, 15, 7, 6, 12, 32, 25 ii) 14, 5, 6, 2, 18, 20, 16, -1, 21
16. Explain and write the analysis of searching and inserting a node in BST.

References

1. Fundamentals of Data Structures in C - Ellis Horowitz and Sartaj Sahni, 2nd edition, Universities Press,2014
2. Data Structures - Seymour Lipschutz, Schaum's Outlines, Revised 1st edition, McGraw Hill, 2014