CHAPTER 8

TREE

"Computers do not solve problems, they execute solutions." ***-Laurent Gasser***

KEY FEATURES

- Binary Tree
- Binary Search Tree
- Threaded Tree
- Expression Tree
- AVL Tree
- Multi-way Search Tree
- B-tree

Tree is one of the most important non-linear data structure in computer algorithms. The drawbacks of linked list can be overcome by using a tree. Many real life problems can be represented and solved using trees. Trees are very flexible, versatile and powerful non-linear data structure that can be used to represent data items possessing a hierarchical relationship among the nodes of the tree.

Definition: A **Tree** may be defined as a non-empty finite set of nodes, such that,

i) There is a specially designated node called the root,

ii) The remaining nodes are partitioned into zero or more disjoint trees T_1, T_2 …T_n are called the **subtrees** of the root R.

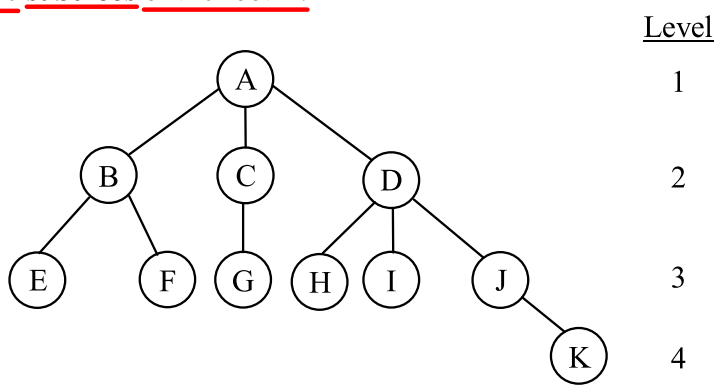


Figure 8.1: A Tree

Terminology of Tree

Node (or vertex): A node stands for the item of information with the branches to other items. Consider the tree in the above figure. This tree has 11 nodes (A, B, C, D, E, F, G, H, I, J and K).

Root: A node without any parent is called root node. In the above figure, A is the root node.

Parent node (or predecessor): Suppose N is a node in a tree with successors s_1, s_2… s_n then N is called the **parent** (or predecessor) of the successors. Each node in the tree, except the root, has a unique parent. In the above figure, node D is the parent of H, I and J.

Children: The successors are called **children** of N. The left and rght successors of node N are called left child and right child of node N respectively. In the above figure, H, J and I are children of node D.

Siblings: The children (or the nodes) of the same parent are said to be **siblings**. In the above figure, H, I and J are siblings.

The degree of a node: The number of subtrees (or children) of a node is called its degree. In the above figure, the degree of node A and D are 3, the degree of B is 2, degree of C is 1, the degree of E, F, G, H, I and K are zero.

The degree of tree: The degree of a tree is the maximum degree of the nodes in the tree. In the above figure, degree 3 is the maximum. Therefore, the degree of the tree is 3.

Internal node (or non-terminal node): The node with at least one child is called internal nodes. In the above figure, A, B, C, D and J are internal nodes.

External nodes (or leaf node): The nodes that have degree zero are called external node or leaf or terminal nodes. In the above figure, E, F, G, H, I and K are the leaf nodes.

Level: The level of a node is defined as follows:

i) The root of the tree is at level one.
ii) If a node is at level L, then its children are at level L + 1.

Note: In some literature, the level of a node is defined in such a way that the root of the tree is at level zero.

In the above figure, the corresponding levels are shown; node A is at level 1, nodes B, C, and D are at level 2, nodes E, F, G, H, I and J are at level 3 and node K at level 4.

Height (or depth): The height or depth of a tree is defined to be the maximum level of any node in the tree. In the above figure, the height of the tree is 4.

Forest: A forest is a set of zero or more disjoint trees. The removal of the root node from a tree results in a forest.

Descendant: A node M in the tree is called a descendant of another node N, if M reachable from the N by repeated proceeding from parent to child.

Ancestor and Descendant: A node N is called an ancestor of node M if N is either the parent of M or the parent of some ancestor of M, i.e., there is a succession of children from N to M; the node M is called descendant of the node N. In the above figure, node D is the ancestor of H, I, J and K.

Edge: The line from a node N in the tree to a successor is called an edge**.**

Path and path length: A sequence of consecutive edges from the source node to the destination node is called a path. The number of edges in a path is path length. In the above figure, A→D→J is a path and the path length is 2.

Internal path length: The sum of the levels of all the internal nodes in the tree is called internal path length.

External path length: The sum of the levels of all the external nodes in the tree is called external path length.

Branch: A path ending in a leaf node is called a branch. In the above figure, A→D→J→K is a branch.

Binary Tree

A binary tree is a finite set of nodes, which is either empty or consists of a root and two disjoint binary trees called left subtree and the right subtree. In a binary tree, the degree of any node is either zero, one or two. A binary tree is a special case of an ordered k-ary tree, where k is 2.

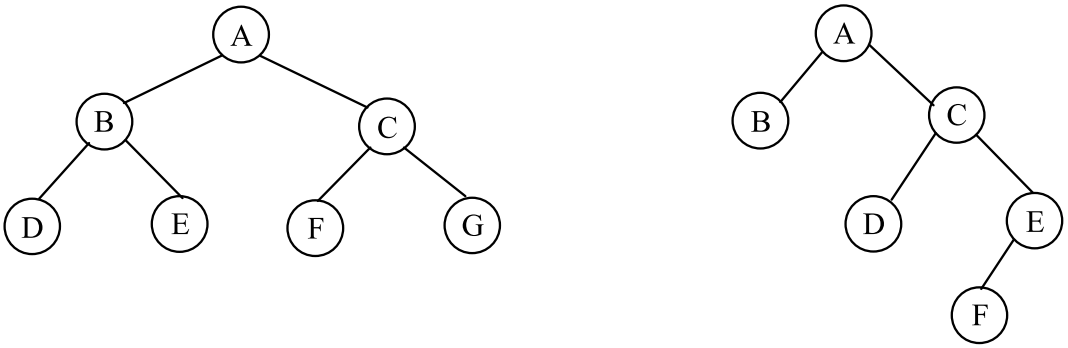


Figure 8.2: Some Binary Trees

Different Types of Binary Trees

Different types of binary trees are possible. Following are common types of binary trees:

- Strictly Binary Tree
- Extended Binary Tree
- Complete Binary Tree
- Full Binary Tree
- Skewed Binary Tree
- Binary Expression Tree
- Balanced Binary Tree
- Threaded Binary Tree
- Binary Search Trees

Strictly Binary Tree

A binary tree is called strictly binary tree if every non-terminal node has non-empty left and right subtree. All the non-terminals nodes must always have exactly two non-empty children. Strictly Binary tree is also known as a 2-ary tree. In the strictly binary tree, the degree of any node is either zero or two, never degree one. A strictly binary tree with n leaves always contains 2n–1 nodes.

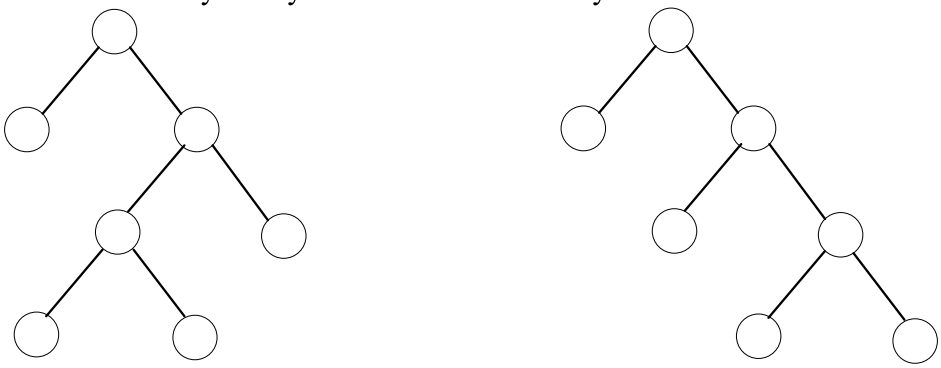


Figure 8.3: Strictly binary trees

Extended Binary Tree

A binary tree can be converted to an extended binary tree by adding special nodes to its leaf nodes and the nodes that have only one child. The extended binary tree also is known as 2-tree. The nodes of the original tree are called internal nodes and the special nodes that are added to the binary tree are called external nodes.

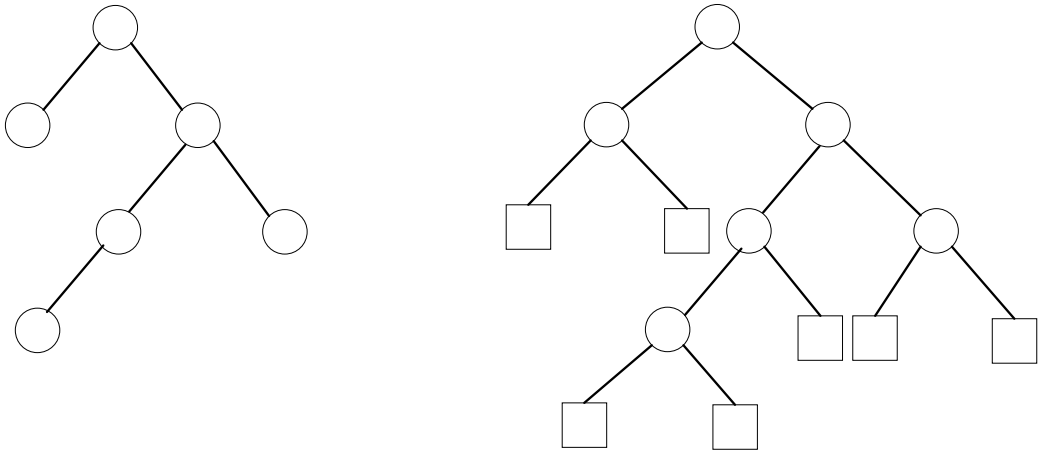


Figure 8.4: Binary tree and its corresponding extended binary tree

Now, we can define the external path length of a binary tree is the sum of all external nodes of the lengths of the paths from the root to those nodes. For example, the external path length E is:

E = 2 + 2 + 4 + 4 + 3 + 3 + 3 = 21

The internal path length is defined as the sum of all internal nodes of the lengths of the paths from the root to those nodes. For example, the internal path length I is:

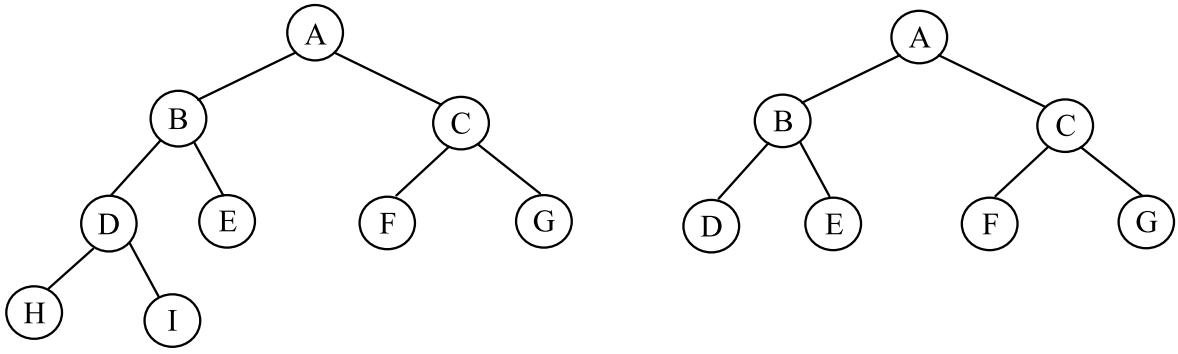
I = 0 + 1 + 1 + 2 + 2 + 3 = 9

Complete Binary Tree

A binary tree is called a complete binary tree in which all the levels are filled and the last level possibly be partially filled from left to right and some rightmost leaves may be missing. The complete binary tree is maximally space efficient. The number of internal nodes in a complete binary tree of n nodes is $\lfloor n/2 \rfloor$. Consider the figure, all the terminal nodes are at the adjacent levels. Practical example of a complete binary tree is Heap.

Full Binary Tree

A binary tree of depth k>=1, contains 2^k-1 nodes, is called full binary tree or perfect binary tree. Note that, a binary tree can have the maximum number of nodes 2^k-1. In the other word, in a full binary tree, all the internal nodes have two children and all the leaves are at the same level. Consider the figure, a full binary tree of depth 3. All full binary tree is a complete binary tree, but all complete binary trees are not full binary tree.



(a) (b)

Figure 8.5: (a) Complete Binary Tree, (b) Full Binary Tree

Skewed Binary Tree

A binary tree, which is dominated solely by left child nodes or right child nodes, is called a skewed binary tree. A left-skewed binary tree has only left child nodes and a right-skewed binary tree has only right child nodes. The height of a skewed binary tree of n nodes is n. In the figure 8.6, the first is a left-skewed binary tree, skewed to left and the second is a right-skewed binary tree, skewed to right. Skewed binary trees are performed worst in all types of trees; such trees are performed similarly as linked list.

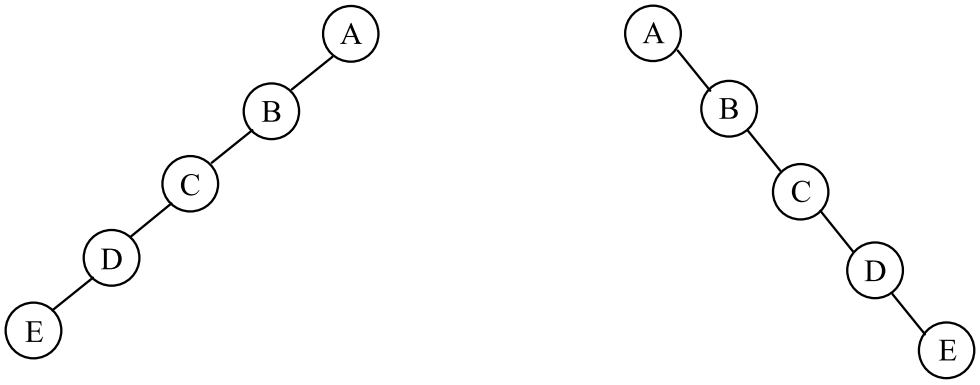


Figure 8.6: Left skewed Binary Tree and Right skewed Binary Tree

Binary Expression Tree

A binary expression tree is a strictly binary tree, which is used to represent a mathematical expression. Two common types of expressions that an expression tree can represent are algebraic and boolean.

This is not necessary that expression tree is always a binary tree. In this example, there are all the binary operators. Therefore, this tree is a binary tree. Although, it is possible to a node to have one child as in the case of a unary operator.

Applications of expression trees:

- Evaluation of expression
- Performing symbolic mathematical operations (such as differentiation) on the expression.
- Generating correct compiler code to actually compute the expression's value at execution time.

Properties of expression tree:

- Expression tree does not contain parenthesis.
- The leaf nodes contain the operands such as constants or variables.
- The non-leaf nodes contain the operators.

Inorder traversal of the expression tree produces infix form of the expression without parenthesis. When the expression tree is traversed in preorder, the prefix (polish) form of the expression is obtained. Similarly, when the expression tree is traversed in postorder then the postfix (reverse polish) form of the expression is obtained. A prefix or postfix form corresponds to exactly one expression tree.

An infix form may correspond to more than one expression tree. Therefore, it is not suitable for expression evaluation.

Consider the infix expression A + B * C. The expression is ambiguous because it produces more

than one expression trees with same infix form.

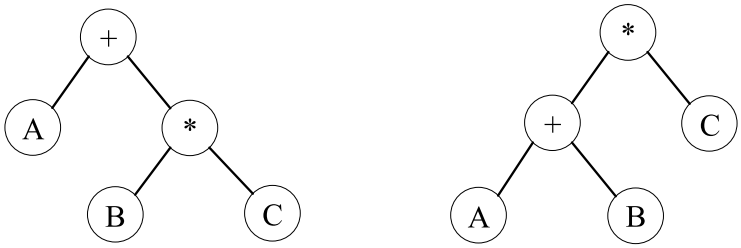


Figure 8.7: Expression Tree of infix expression A + B * C

Table 8.1: Different forms of expression

Expression	Prefix form	Infix form	Postfix form
A + (B * C)	+ A*BC	A + B * C	ABC*+
(A + B) * C	*+ABC	A + B * C	AB+C*

Construction of Expression Tree

An expression can be converted to it equivalent postfix expression. The following algorithm constructed an expression tree from a valid postfix expression P containing binary operators:

Algorithm to create expression tree

```
Algorithm: CREATE_EXPRESSION_TREE (P)
1. Repeat while not the end of the expression P
2.    Read the postfix expression one symbol S at a time
3.    If S is an operand then
         i) Create a node for the operand
        ii) Push the pointer to the created node onto a stack
4.    If S is a binary operator then
         i) Create a node for the operator
        ii) T1 = Pop from the stack a pointer to an operand
       iii) T2 = Pop from the stack a pointer to an operand
        iv) Make T2 the left subtree and T1 the right subtree of the
            operator node
         v) Push the pointer to the operator node onto the stack
    [End of while]
5. T = Pop from the stack a pointer to expression tree
6. Return
```

Example:

Draw a expression tree from the following expression:

A * B ^ C / D + E

Now, at first converts the above expression to its equivalent postfix expression: A B C ^ * D / E +

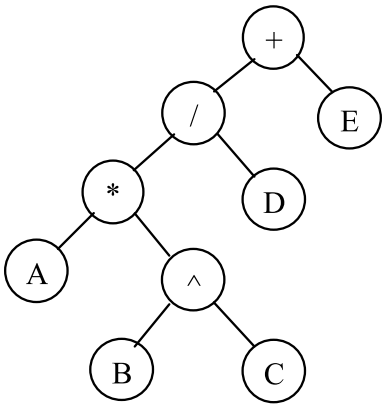


Figure 8.8: Expression Tree

The preorder traversal of the above expression tree is + / * A ^ B C D E, this is the prefix form of the expression. The postorder traversal of the above expression tree is A B C ^ * D / E +, this is the postfix form of the expression.

Evaluation of Expression Tree

A postfix expression can be converted to expression tree using the above algorithm. Now we can evaluate the expression tree, using the following algorithm:

Algorithm to evaluation of expression tree

```
Algorithm: EVALUATION (T)
1. If T is a leaf then
2.    Return value of T's operand
3. Else
4.    Operator = T.Element
5.    Operand1 = EVALUATION(T.Left)
6.    Operand2 = EVALUATION(T.Right)
7.    Return (Operation (Operand1, Operator, Operand2))
   [End of If]
8. End
```

Balanced Binary Tree

There are mainly two types of balanced binary trees.

i) Weight balanced binary tree
ii) Height balanced binary tree

Weight Balanced Binary Tree

A weight-balanced binary tree is a binary search tree if for each node it holds that the number of inner nodes in the left subtree and the number of inner nodes in the right subtree differ by at most one. These trees can be used to implement dynamic sets, dictionaries (maps) and sequences. The weight-balanced binary trees were introduced by Nievergelt and Reingold in 1972. It is purely functional implementations are widely used in functional programming languages.

The balance of weight-balanced binary tree is based on the sizes (number of elements) of the subtrees in each node. The size of the leaf node is zero. The size of the internal nodes is the sum of sizes of its two children, plus one (size [n] = size [n.left] + size [n.right] + 1). Based on the size, one defines the weight as either equal to the size, or as weight [n] = size [n] + 1. Now, insertion and deletion operations that modify the tree must make sure that the weight of the left and right subtrees of every node remain within some factor α of each other.

Height Balanced Binary Tree

A height-balanced binary tree has the minimum height for the leaf nodes. A binary tree is height balanced if height of the tree is O ($\log n$) where n is number of nodes. One common height-balanced tree structure is a binary tree structure in which the left and right subtrees of every node differ in height by no more than one.

For Example, AVL tree (Adel'son-Vel'skii and E. M. Landis, 1962) maintain O ($\log n$) height by making sure that the difference between heights of left and right subtrees is at most ±1. Red-Black trees (Guibas and Sedgewick, 1978) maintain O ($\log n$) height by making sure that the number of Black nodes on every root to leaf paths are same and there are no adjacent red nodes. Balanced Binary Search trees are performed good as they provide O ($\log n$) time for search, insert and delete operations.

Threaded Binary Tree

A threaded binary tree is a binary tree in which having a loop. In a binary tree, most of the entries in the link field will contain null elements. These null entries are replaced by special pointers, which point to nodes higher in the tree. These special pointers are called threads and binary trees with such pointers are called threaded tree.

For a n-node binary tree, there exists

- total 2n number of pointers or link fields
- total n-1 number of actual (not null) pointers or link fields
- total n+1 number of null pointers or link fields

For optimizing null pointers, the concept of the thread is used. There are different types of threaded binary trees are possible, inorder threaded binary trees, preorder threaded binary trees, and postorder threaded binary trees correspond to inorder, preorder and postorder traversals. Each type of threaded binary trees can be of two representations: one-way threading and two-way threading.

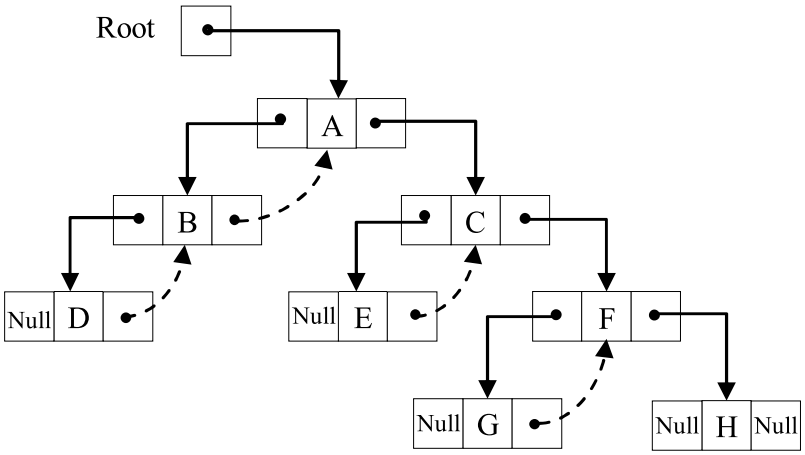


Figure 8.9: One-way inorder threaded binary tree

In the one-way inorder threaded binary tree, an only right null pointer used as a thread that will point to the next node in the inorder traversal (i.e. inorder successor). However, the right null pointer of the last node remains unused. In the two-way inorder threaded binary tree, left null pointer acts as a thread that will point to the previous node in the inorder traversal (i.e. inorder predecessor) and right null pointer acts as a thread that will point to the next node in the inorder traversal (i.e. inorder successor). However, the left null pointer of the first node and the right null pointer of the last node remains unused.

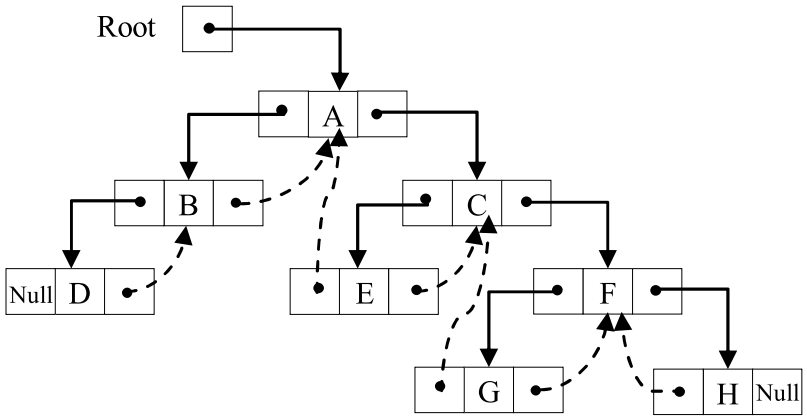


Figure 8.10: Two-way inorder threaded binary tree

The structure definition for a node of two-way inorder threaded binary tree as follows:

```
struct Node
{
   struct Node *left;
   char lthread;
   int info;
   struct Node *right;
   char rthread;
};
typedef struct Node ThreadedTreeNode;
```

Properties of Binary Tree

Lemma 1: A binary tree with n nodes has exactly n – 1 edges (same as any normal tree).

Proof: The property can be proof by induction.

Induction Base: Let n = 1. That is there is only one node in the tree. Therefore, a number of the edge is 0. Hence, the property is true for one node.

Induction Hypothesis: Assume the property is true for n = k i.e., for k nodes there is k – 1 edges.

Induction Step: The number of edges for k nodes is k - 1 edges by the induction hypothesis. Now addition of one node (i.e. n = k + 1) includes one extra edge. Therefore, total number of edges are = (k – 1) + 1 = k. Hence proved.

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

Proof: The property can be proof by induction on i.

Induction Base: The root is the only node on level i=1.

Hence the maximum number of nodes on level = 1 is $2^0 = 2^{i-1}$.

Induction Hypothesis: For all j, $i \leq j < i$, the maximum number of nodes on level j is 2^{j-1}.

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 maximum degree 2, the maximum number of nodes on level i is two times the maximum number on level i-1, i.e. $2 \times 2^{i-2} = 2^{i-1}$.

Lemma 3: The maximum number of nodes in a binary tree of depth k is $2^i - 1$, $k \geq 1$.

Proof: The maximum number of nodes on level i of a binary tree is 2^{i-1}. Therefore, the maximum number of nodes in a binary tree of depth k is,

$$= \sum_{i=1}^{k} (\textbf{maximum number of nodes on level i})$$

$$= \sum_{i=1}^{k} (2^{i-1})$$

$$= 2^k - 1$$

Lemma 4: For any non-empty binary tree T, if n_0 is the number of leaves (terminal nodes) and n_2 be the number of nodes having degree 2 then $n_0 = n_2 + 1$.

Proof: Let n_1 be the number of nodes of degree 1 and n is the total number of nodes. Since all the nodes in T are of degree ≤ 2 we have,

$$n = n_0 + n_1 + n_2 \qquad \text{(i)}$$

Now, if we count the number of branches in a binary tree, we see that every node except for the root has a branch leading into it. If B is the number of branches, then n = B + 1. All the branches come either from a node of degree one or from a node of degree two. Thus, $B = n_1 + 2n_2$. Hence, we obtain

$$n = 1 + n_1 + 2n_2 \qquad \text{(ii)}$$

Subtracting (ii) from (i) and rearranging terms we get

$$n_0 = n_2 + 1$$

Lemma 5: If n is the total number of nodes in a complete binary tree of height h, then $h = \lfloor \log_2 n \rfloor + 1$.

Proof: From the definition of a complete binary tree of height h, it is filled up to height h – 1 and in the last level, it may have partially filled with nodes. Hence, we can write:

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

Since the maximum number of nodes at height h-1 is $2^{h-1} - 1$ and at height, h is $2^h - 1$.

or we can write $2^{h-1} \leq n < 2^h$ (iii)

Taking the logarithm of (iii) we get

$h - 1 \log_2 n < h$

Therefore, the value of $\log_2 n$ lies between h and h-1. Now if we take floor value of $\log_2 n$ then it will be h-1.

Hence, $h = \lfloor \log_2 n \rfloor + 1$ or $h \leq \lfloor \log_2 n + 1 \rfloor$

Representation of Binary Tree

Tree is a widely used abstract data type since it is defined in terms of operations on it and its implementation is hidden. Therefore, we can implement a tree using either array or linked list. Binary Tree also can be representation with two different ways.

i) Using array

ii) Using linked list

Binary Tree Representation with Array

Binary Tree can be represented by the array. There are two different ways to represent a binary tree with array.

- Linked Representation
- Sequential Representation

Linked Representation

In Linked representation, a Binary Tree can be stored in computer memory by using three parallel arrays, DATA, LCHILD and RCHILD and a pointer variable ROOT. Now, each node N of binary tree T will correspond to a location k such that,

i) DATA[K] contains the data at the node N

ii) LCHILD[K] contains the location of the left child of node N

iii) RCHILD[K] contains the location of the right child of node N

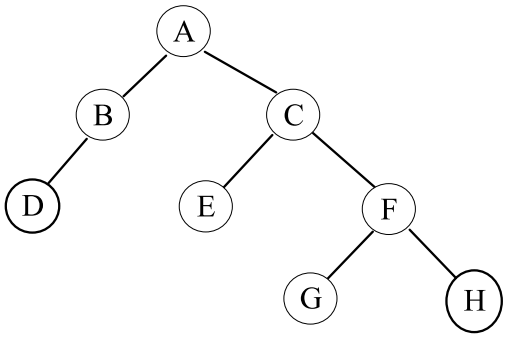


Figure 8.11: A Binary Tree

Root

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
DATA	E		A		G	C		B		F		D			H
LCHILD	0		8		0	1		12		5		0			0
RCHILD	0		6		0	10		0		15		0			0

Figure 4.12: Linked Representation of above Binary Tree

Sequential Representation

There is an efficient way of maintaining or to store a Binary Tree in computer memory when the tree is complete or nearly complete. Since heap is a complete binary tree, therefore a heap can also be represented by this representation.

In the sequential representation, a Binary Tree T can be represented by using only a single linear array TREE, such that

i) The root of T is stored in TREE[1].

ii) When a node N stores in TREE[K], then its left child will be stored in TREE[2*K] and right child will be stored in TREE[2*K+1].

Therefore, if K is the index of current node, then parent node is stored in the FLOOR (K/2).

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D		E	F							G	H

Figure 8.13: Sequential Representation of above Binary Tree

For a zero-based array,

i) The root of T is stored in TREE[0]

ii) When a node N stores in TREE[K], then its left child will be stored in TREE[2*K+1] and right child will be stored in TREE[2*K+2].

Therefore, if K is the index of a current node then parent node is stored in FLOOR ((K-1)/2).

For a complete binary tree, sequential representation is perfect as no space is wasted. However, it is wasteful for many other binary trees. For a skewed binary tree, less than half the array can be utilized. In the worst case, a skewed binary tree of depth k will require 2^k-1 memory space. In addition, insertion or deletion of a node in the middle of the tree requires movement of many nodes. These problems can be overcome using linked list representation.

Threaded Binary Tree Representation with Array

Threaded Binary Tree T may be stored in computer memory by using a linked representation. Here, the thread can be represented by a negative value of the location and ordinary pointer can be represented by the positive value of the location.

Root

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
DATA	E		A		G	C		B		F		D			H
LCHILD	−3		8		−6	1		12		5		0			−10
RCHILD	−6		6		−10	10		−3		15		−8			0

Figure 8.14: Linked Representation of above Two-way Inorder Threaded Binary Tree

Binary Tree Representation with Linked List

Binary Tree can be also representation with Linked List. In this representation, each node of a binary tree consists three fields such that

- The first field contains the pointer field, which points to the left child.
- The second field contains the data.
- The third field contains the pointer field, which points to the right child.

LChild	Data	RChild

The structure definition for a node of binary tree as follows:

```
struct Node
{
   struct Node *left;
   int info;
   struct Node *right;
};
typedef struct Node BTreeNode;
```

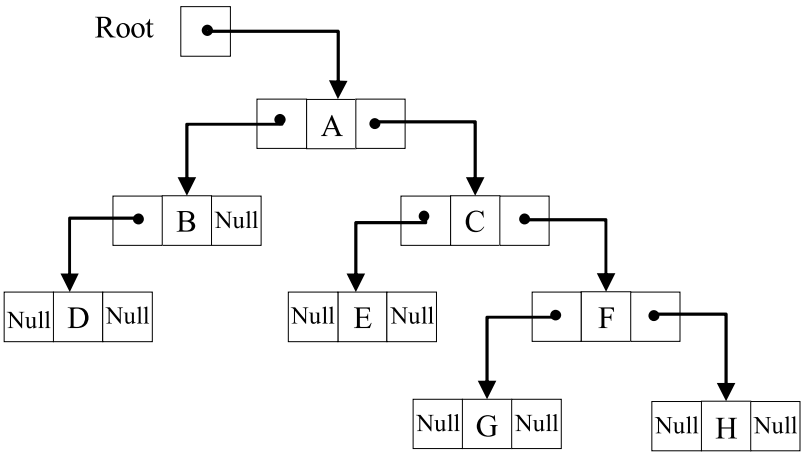


Figure 8.15: Linked Representation of Binary Tree

Binary Tree Traversal

Tree traversal is the operation of visiting each node in the tree exactly once. There are mainly two traversals are:

i) Depth First Search (DFS)
ii) Breadth First Search (BFS)

Depth First Search

Depth first search traversal can be implemented easily using a stack, including recursion. Starting from the root of the binary tree, there are three main steps that can be performed. These steps are moving left (L), visiting node (D) and moving right (R), then there are six possible combinations of traversal: LDR, LRD, DLR, DRL, RDL and RLD. Now if we traverse left before right then only three traversal remains: LDR, LRD and DLR; LDR is known as inorder, LRD is postorder and DLR is preorder, these three are only standard binary tree traversals.

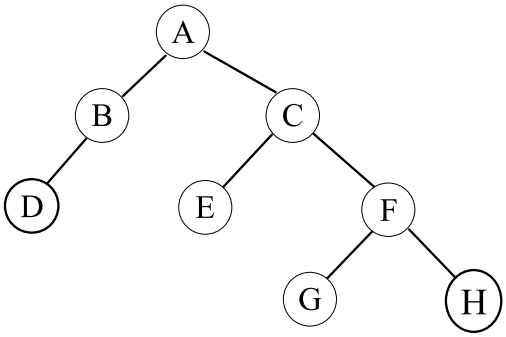


Figure 8.16: Binary Tree

Inorder Traversal

The inorder traversal of a non-empty binary tree is defined as follows, starting from root node:

i) Traverse the left subtree of root in inorder.
ii) Visit the root node.
iii) Traverse the right subtree of the root in inorder.

Algorithm finds the inorder traversal of a binary tree using recursion

```
Algorithm: INORDER (ROOT)
1. IF ROOT ≠ NULL THEN
       a) INORDER(ROOT → LCHILD)
       b) PRINT: ROOT → INFO
       c) INORDER(ROOT → RCHILD)
2. RETURN
```

The inorder of the above binary tree is:

D B A E C G F H

Algorithm finds the inorder traversal of a binary tree in a non-recursive mode.

```
Algorithm: INORDER (ROOT)
1. P = ROOT
2. Initialize Stack
3. Repeat while Stack is not empty or P ≠ null
4.     Repeat while P ≠ null
       a) PUSH(Stack, P)
       b)P = P → LCHILD
       [End of loop]
5.     If Stack is not empty then
       a) P = POP(Stack)
       b)Print: P → Info
       c)P = P → RCHILD
   [End of loop]
6. Return
```

Preorder Traversal

The preorder traversal of a non-empty binary tree is defined as follows, starting from root node:

i) Visit the root node.
ii) Traverse the left subtree of root in preorder.
iii) Traverse the right subtree of the root in preorder.

Algorithm finds the preorder traversal of a binary tree using recursion.

```
Algorithm: PREORDER (ROOT)
1. IF ROOT ≠ NULL THEN
        a) PRINT: ROOT → INFO
        b) PREORDER(ROOT → LCHILD)
        c) PREORDER(ROOT → RCHILD)
2. RETURN
```

The preorder of the above binary tree is:

A B D C E F G H

Algorithm finds the preorder traversal of a binary tree in a non-recursive way

```
Algorithm: PREORDER (ROOT)
1. Initialize Stack
2. PUSH(Stack, ROOT)
3. Repeat while Stack is not empty
      a)P = Pop(Stack)
      b)If P ≠ null then
           i) Print: P → Info
          ii) Push(Stack, P → RCHILD)
         iii) Push(Stack, P → LCHILD)
   [End of loop]
4. Return
```

Postorder Traversal

The postorder traversal of a non-empty binary tree is defined as follows, starting from root node:

i) Traverse the left subtree of root in postorder.
ii) Traverse the right subtree of the root in postorder.
iii) Visit the root node.

Algorithm finds the postorder traversal of a binary tree using recursion.

```
Algorithm: POSTORDER (ROOT)
1. IF ROOT ≠ NULL THEN
      a) POSTORDER(ROOT → LCHILD)
      b) POSTORDER(ROOT → RCHILD)
      c) PRINT: ROOT → INFO
2. RETURN
```

The preorder of the above binary tree is:
D B E G H F C A

Algorithm finds the postorder traversal of a binary tree in a non-recursive way.

```
Algorithm: POSTORDER (ROOT)
1.  Initialize Stack
2.  P = ROOT
3.  Repeat while Stack is not empty
4.    Repeat while P ≠ null
5.       Push(Stack, P)
6.       If P → RCHILD ≠ null
7.          Push(Stack, null)
8.       P = P → LCHILD
      [End of while]
9.  Q = Pop(Stack)
```

```
10. If Q ≠ null
11.     PRINT: Q → INFO
12. Else
13.     Q = Pop(Stack)
14.     P = Q → RCHILD
15.     Push(Stack, Q)
    [End of If]
    [End of loop]
16. Return
```

Breadth First Search

Binary trees can also be traversed in level-order, where every node are visited on a level before the next level. This search is known as Breadth First Search (BFS). Breadth first search traversal can be implemented easily using a queue.

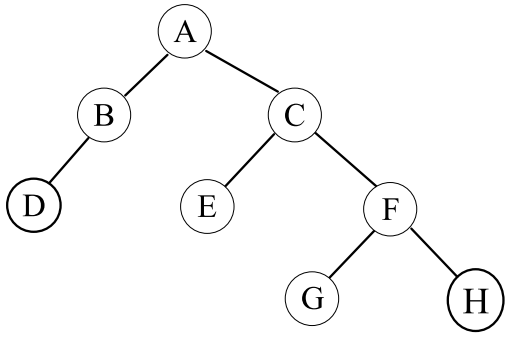


Figure 8.17: A Binary Tree

The breadth first search of the above binary tree is:

A B C D E F G H

Reconstruction Binary Tree from its Traversals

An original tree cannot be reconstructed given by its inorder or preorder or postorder traversal alone. However, a unique binary tree can be reconstructed either by inorder and preorder traversals, or by inorder and postorder traversals. However, preorder and postorder traversals give some ambiguity in the tree structure.

- The first node visited in a preorder traversal of a binary tree is the root, and then left subtree and right subtree are traversed.
- In postorder traversal of a binary tree, left subtree and right subtree are traversed then the last node visited is the root.
- In inorder traversal of a binary tree, left subtree is traversed first, then the root node is visited, finally right subtree is traversed.

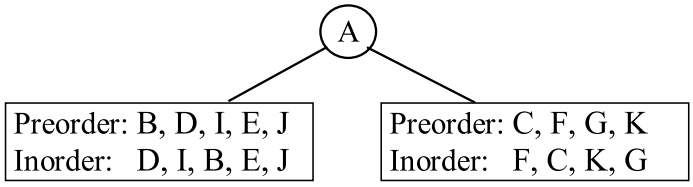
Example:

The inorder and preorder traversal sequence of nodes in a binary tree are given below:

Preorder: A, B, D, I, E, J, C, F, G, K
Inorder : D, I, B, E, J, A, F, C, K, G

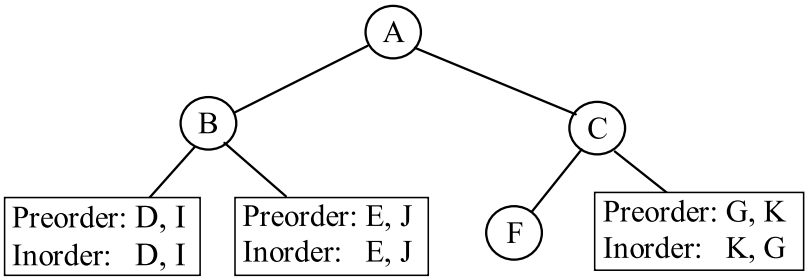
The following steps are used to reconstruct the binary tree:

In preorder traversal of a binary tree, the root node is visited first and then left subtree is traversed, and finally, the right subtree is traversed. In inorder traversal of a binary tree, left subtree is traversed first, then the root node is visited, finally, the right subtree is traversed. Therefore, in the preorder traversal, the first node A must be root. Now by searching node A in inorder traversal we can find out all elements on the left side of A are an inorder traversal of left subtree and elements on right are an inorder traversal of right subtree. Therefore, inorder and preorder traversal of left subtree and right subtree can be obtained.



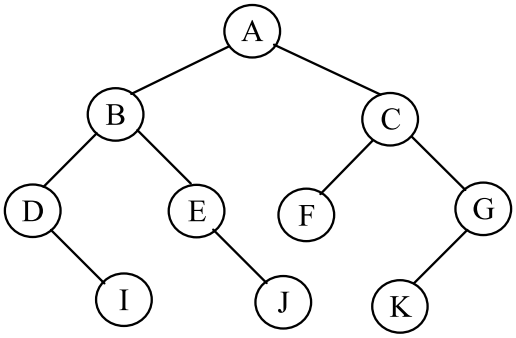
(a)

Similarly, node B is the root of the left subtree and node C is the root of the right subtree. Inorder and preorder traversal of left subtree and right subtree of B and C can be found. The node F is the obviously left child of node C.



(b)

Similarly, node D and E are the roots of the left subtree and the right subtree of node B. the node G is the right subtree of node C. In addition, node I is the right child of node D, node J is the right child of node E and node K is the left child of node K.



(c)

Figure 8.18 (a, b, c): Reconstruction of Binary Tree

Binary Search Tree

A Binary search tree (BST) is an ordered binary tree. P. F. Windley, A. D. Booth, A. J. T. Colin and T. N. Hibbard invent binary search tree, in 1960.

Definition: A Binary Search Tree, which is either empty or each node in the tree contains a key and

i) All keys in the left subtree are less than the keys in the root node,
ii) All keys in the right subtree are greater than the keys in the root node,
iii) The left and right subtrees are also binary search tree.

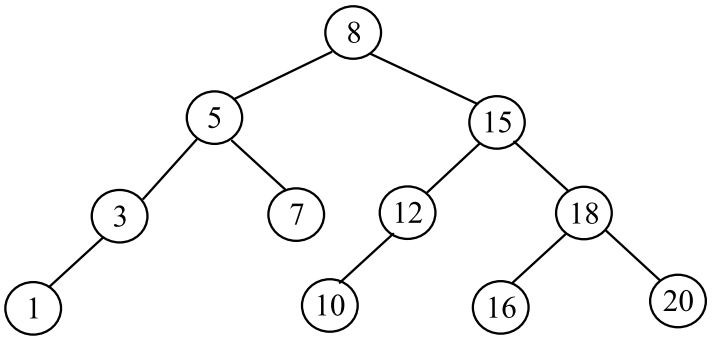


Figure 8.19: Binary Search Tree

Operations on Binary Search Tree

Operations supported by a Binary search tree are as follows:

Table 8.2: Various Operation on Binary Search Tree

Operation	Description
Traverse	This operation traversing all the nodes of binary search tree exactly once.
Insertion	This operation insert a node in the binary search tree
Deletion	This operation removes a node from the binary search tree
Searching	This operation searches a desired key value within the binary search tree.
Successor	This operation finds the successor of a given node in the binary search tree.
Predecessor	This operation finds the predecessor of given node in the binary search tree.

Binary Search Tree Traversal

The tree traversal algorithm (preorder, postorder and inorder) are the standard way of traversing a binary search tree, which is similar as traversing in a binary tree. In a binary search tree, inorder traversal always retrieves data items in increasing sorted order.

Insertion in Binary Search Tree

Suppose a new data item having a key and the tree in which the key is inserted are given as an input. Insertion operation starts from the root node. If the tree is empty then the new item inserted as the root node. Otherwise, if the tree is non-empty then compare the value of the key with the root node. If the key is less than the root node then it is inserted in the left subtree, otherwise it is inserted in the right subtree.

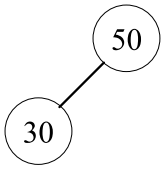
Example:

Insert the keys in the Binary Search Tree: 50, 30, 10, 90, 100, 40, 60, 20, 110, 5

Insert 50:

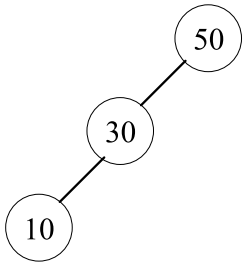
(a)

Insert 30:



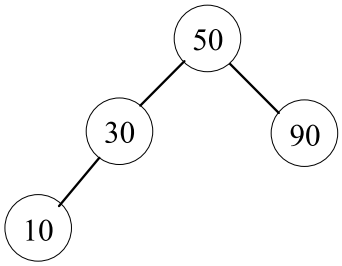
(b)

Insert 10:



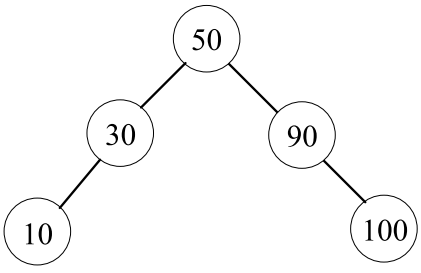
(c)

Insert 90:



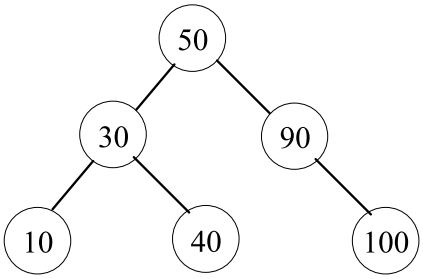
(d)

Insert 100:



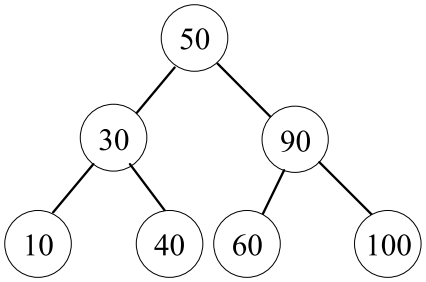
(e)

Insert 40:



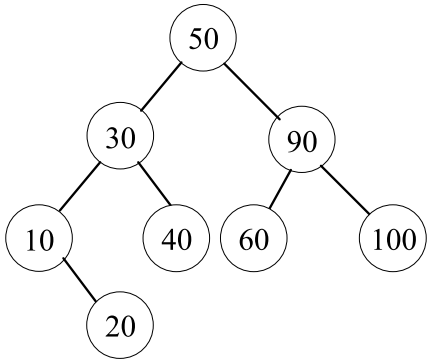
(f)

Insert 60:



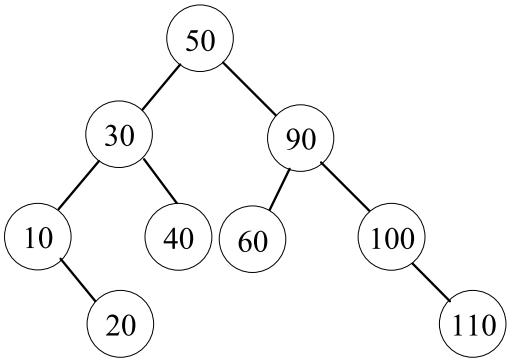
(g)

Insert 20:



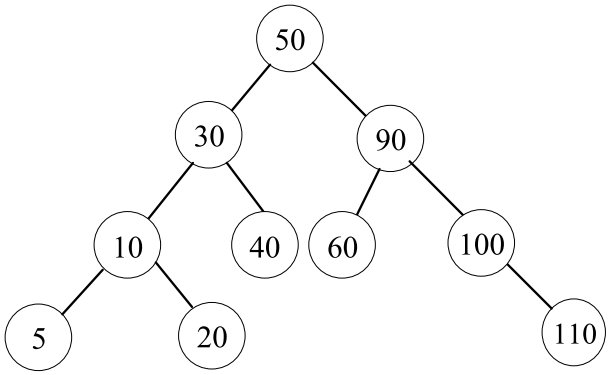
(h)

Insert 110:



(i)

Insert 5:



(j)

Figure 8.20(a-j): Insertion in Binary Search Tree

Algorithm to insert an item to binary search tree using recursion method

```
Algorithm: INSERT (ROOT, DATA)
1. IF ROOT = NULL THEN
       i) Allocate Memory for ROOT node.
       ii) ROOT->INFO=DATA
       iii) ROOT->LCHILD=NULL
       iv) ROOT->RCHILD=NULL
2. ELSE IF ROOT->INFO>DATA THEN
       CALL INSERT(ROOT->LCHILD, DATA)
3. ELSE IF ROOT->INFO<DATA THEN
       CALL INSERT (ROOT->RCHILD,DATA)
4. RETURN
```

Searching in Binary Search Tree

Similar to traversing, insertion operations in the binary search tree, search algorithm also utilized the recursion technique.

Suppose a key and the tree in which the key is searched for are given as an input. Now starting from the root node, check whether the value of the current node equals to the key or not. In the case, when a current node is null then the searched key value does not exist in the binary search tree. If the node has the key that is being searched for, then the search is successful.

Otherwise, the key of the current node is either smaller than or greater than the searching key value. In the first case, all the keys in the left subtree are less than the searching key value. That means do not need to search in the left subtree. Thus, it needs to search only the right subtree. Similarly, in the second case, it needs to search only the right subtree.

Algorithm to search an item from a binary search tree using recursion

```
Algorithm: BSTSearch (ROOT, DATA, P)
1. IF ROOT = NULL THEN
       i) PRINT: NOT FOUND
       ii) P = NULL
       iii) RETURN
2. IF ROOT->INFO=DATA THEN
       SET P=ROOT
3. ELSE IF ROOT->INFO>DATA THEN
       CALL BSTSearch(ROOT->LCHILD,DATA,P)
4. ELSE CALL BSTSearch(ROOT->RCHILD,DATA,P)
5. RETURN
```

Algorithm to search an item from a binary search tree using iterative methods

```
Algorithm: BSTSearch (ROOT, DATA, P)
1. P = ROOT
2. Repeat while P ≠ Null
```

```
3.    If DATA = P->INFO then Return
4.    Else If DATA < P->INFO then
5.          P = P->LCHILD
6.    Else P = P->RCHILD
   [End of loop]
7. Return
```

Suppose a binary search tree contains n data items. Therefore, there are n! Permutations of the n items. The average depth of the n! numbers of the tree is approximately $c\log_2 n$, where c = 1.4. The average running time f(n) to search for an item in a binary search tree with n elements is proportional to $\log_2 n$, that is f (n) = O($\log n$).

Inorder Successor of a Node

In a binary tree, inorder seccessor of a node is the next node in inorder traversal of the binary tree. Inorder successor is NULL for the last node in inorder traversal. In binary search tree, inorder successor of a node with key k is a smallest key value that belongs to the tree and that is strictly greater than k. The idea for finding the successor of a given node x:

i) If the x has a right child then its inorder successor will the left most element in the right suB-tree of x (i.e. the minimum in the right subtree of x).

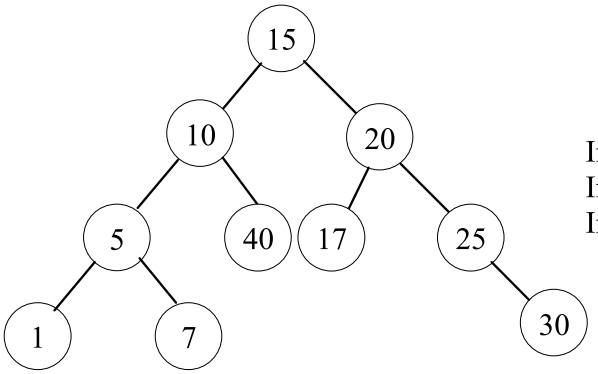
ii) Otherwise, if the x doesn‘t have a right child then its inorder successor will the one of its ancestors, the inorder successor is the farthest node that can be reached from x by following only right branches backward.

iii) Otherwise, if x is the right most node in the tree then its inorder successor will be NULL.

Algorithm to find the inorder successor from binary search tree

```
Algorithm: BST_SUCCESSOR(X)
1. If X → RCHILD ≠ NULL then
2.    Y = X → RCHILD
3.    Repeat while Y→LCHILD ≠ NULL
4.       Y = Y → LCHILD
5.    [End of loop]
6.    Return Y
7. Else
8.    Y = X
9.    Repeat while Parent(X) → RCHILD = X
10.       X = Parent(X)
     [End of loop]
11.  If Parent(X) ≠ NULL then Return Parent(X)
12.  Else Print: No successor
     [End of If]
   [End of If]
13. Return
```

Example:



Inorder successor of 1 is 5
Inorder successor of 7 is 10
Inorder successor of 15 is 17

Figure 8.21: Binary Search Tree

Deletion in Binary Search Tree

Deletion operation in the binary search tree, there are three possible cases has to consider.

Case 1: When the deleting node with no children or leaf node (i.e. deleted node is a leaf node), then simply remove the node from the tree and set null to the parent's corresponding link.

Case 2: When the deleting node with one child, either left or right child (i.e. deleted node has exactly one non-empty subtree), then simply replace the node with its unique child.

Case 3: when the deleting node (P) with two children (i.e. deleted node has exactly two subtrees), then select its inorder successor node or its inorder predecessor node (R). Copy the key value from node R to node P and then recursively delete the R node until satisfying one of the first two cases. In a binary tree, inorder successor (R) of a node is only its right subtree's left-most child, as right subtree is not null (in the present case the node has two children). Now inorder successor may have zero children or only one right child, therefore it can be deleted using one of the first two cases.

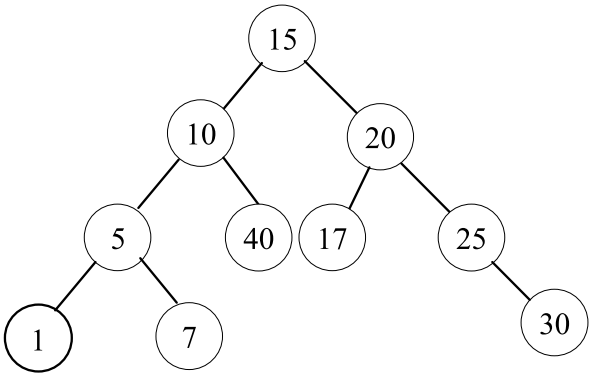
Algorithm to delete an item from binary search tree

```
Algorithm: DELETE (ROOT, P, PARENT, DATA)
1. IF P→LCHILD=NULL AND P→RCHILD=NULL THEN
      i) IF PARENT→LCHILD=P THEN
            SET PARENT→LCHILD=NULL
      ii) ELSE SET PARENT→RCHILD=NULL
2. ELSE IF P→LCHILD=NULL THEN
      i) IF PARENT→LCHILD=P THEN
            SET PARENT→LCHILD=P→RCHILD
      ii) ELSE SET PARENT→RCHILD=P→RCHILD
3. ELSE IF P→RCHILD=NULL THEN
      i) IF PARENT→LCHILD=P THEN
            SET PARENT→LCHILD=P→LCHILD
      ii) ELSE SET PARENT→RCHILD=P→LCHILD
4. ELSE P→LCHILD≠NULL AND P→RCHILD≠NULL THEN
      i) SET PARENT = P
      ii) SET IN=P→RCHILD
```

```
      iii) REPEAT WHILE IN→LCHILD≠ NULL
             a) SET PARENT = IN
             b) SET IN=IN→LCHILD
             [END OF LOOP]
      iv) SET P→INFO=IN→INFO
      v) SET P = IN
      vi) Call DELETE(ROOT, P, PARENT, DATA)
    [END OF IF]
5. Deallocate memory for P node.
6. RETURN.
```

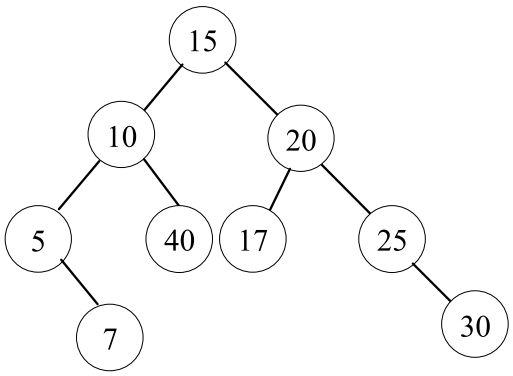
Example:

Delete the keys from the following Binary Search Tree: 1, 30, 5, 15



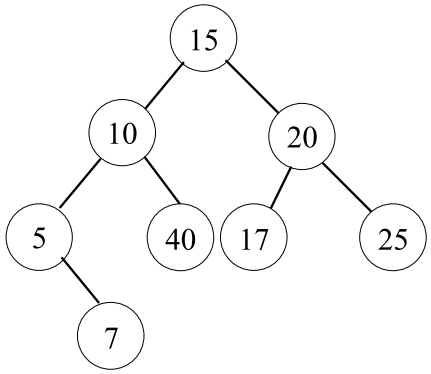
(a)

Delete 1: Here the deleted node is the left child of its parent node. Hence after setting `PARENT→LCHILD=NULL` we get,



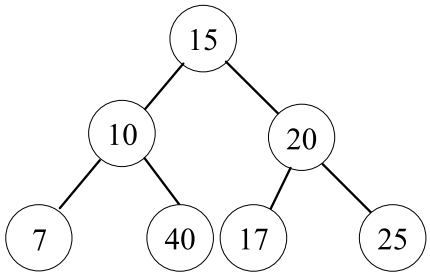
(b)

Delete 30: Here the deleted node is the right child of its parent node. Hence, after setting `PARENT→RCHILD=NULL` we get,



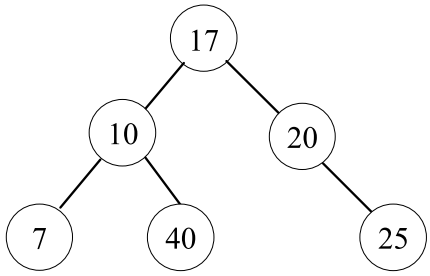
(c)

Delete 5: Here the deleted node is the left child of its parent node and it has a right subtree. Hence, after setting `PARENT→LCHILD=P→RCHILD` we get,



(d)

Delete 15: Here the deleted node has two subtrees. At first, find the inorder successor of the deleted node then substituting the key of the deleted node by the key of its inorder successor. Finally, delete the inorder successor.



(e)

Figure 8.22(a-e): Deletion in Binary Search Tree

On a binary search tree of height h, different operations like search, minimum, maximum, successor, predecessor, insert and delete can be made to run O (h) time. On average, binary search trees with n nodes have $\log n$ height and in the worst-case, binary search trees can have n height. Therefore, different operations like search, minimum, maximum, successor, predecessor, insert and delete take O ($\log n$) time in average-case and O (n) in worst-case.

Binary search trees are a basic data structure used to construct abstract data structures such as sets, multisets, associative arrays. To sort a sequence of numbers, at first, all numbers are required to insert into a new binary search tree then traverse the tree in inorder.

Advantages of Binary Search Tree

The major advantage of binary search tree over other data structures are as follows:

i) Sorting and search algorithm can be very efficient.

ii) Easy to the coding of most of the operations that performed on binary search tree.

Disadvantages of Binary Search Tree

i) The shape of the binary search tree fully depends on the sequence of insertions and deletions operations may result in skewness.

ii) The height of the binary search tree is much higher than $\log n$ in the most of the cases, as a result, runtime may increase.

iii) As the binary search tree is not a balanced tree, run time of most of the operations is O (n) in the worst case.

HEAP

Heap is a binary tree that must satisfy the following properties:

i) The binary tree essentially complete that means the tree completely filled all levels, the last level may be partially filled from left to right, and some rightmost leaves may be missing.

ii) All keys in the tree, other than the root node, are greater/smaller than or equal to the key in the parent node.

There are two types of the heap:

i) Max Heap and ii) Min Heap

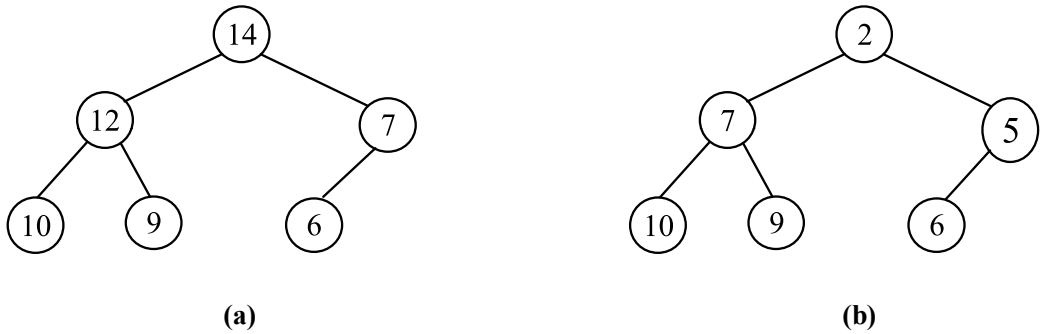


Figure 8.23: (a) Max Heap and (b) Min Heap

Max Heap

A Max Heap is defined to be a complete binary tree with the property that the key of each node is greater than or equal to the keys of its children nodes.

Min Heap

A Min Heap is also a complete binary tree with the property that the key of each node is smaller than or equal to the keys of its children nodes.

Operations on Heap

Operations supported by a Heap are as follow:

Table 8.3: Various Operation on Heap

Operation	Description
Heapify	This operation restores the heap condition. For example, if a node changed in the tree, the heap condition is not valid anymore. Then it needs to restore the condition by moving nodes up or down the tree.
Insertion	This operation inserts a node in the heap.
Deletion	This operation removes a node from the heap.
Shift-up	This operation moves a node up in the tree, as long as needed (depending on the heap condition: min-heap or max-heap).
Shift-down	This operation moves a node down in the tree.

Application of Heap

- Operating Systems- Jobs / Process scheduling
- Heap Sorting
- Graph Application
- Priority Queue

AVL Tree

A binary tree is height balanced binary tree if it is either empty or if T is a non-empty binary search tree with T_L and T_R as its left and right subtrees, if and only if

i) T_L and T_R are height balanced and

ii) $|h_L-h_R| <= 1$ where h_L and h_R are the heights of T_L and T_R respectively.

It is introduced by two Russian mathematicians, **G. M. Adelson-Velskii** and **E. M. Landis**, in 1962. Hence, such trees are known as AVL trees.

The **balance factor** BF (T) of a node T in a binary tree is defined to be $h_L - h_R$ where h_L and h_R are the heights of the left and right subtrees of T. For any node in an AVL tree BF (T) = –1, 0 or 1.

Operations on AVL Tree

Operations supported by an AVL tree are as follow:

Table 8.4: Various Operation on AVL Tree

Operation	Description
Traverse	This operation traversing all the nodes of AVL tree exactly once.
Insertion	This operation inserts a node in the AVL tree
Deletion	This operation removes a node from the AVL tree
Searching	This operation searches a desired key value within the AVL tree.

Insertion in an AVL Tree

When a new node is inserted to a balanced binary search tree, as a result the tree could be unbalanced. The re-balancing was carried out using four different kinds of rotations LL, RR, LR and RL.

These rotations are characterized by nearest ancestor A on the path from the inserted node B to the root node, whose balanced factor becomes ±2.