1. **Tree.** A *tree* is a collection of nodes.
   i. The collection can be empty.
   ii. A tree consists of a distinguished node *r*, called the *root*, and zero or more nonempty subordinate trees *T*₁, *T*₂, ..., *T*ₖ. Each subordinate tree is connected by a directed *edge* from *r* to the root of the subordinate tree.

2. **Child.** The root of each subordinate tree, *T*ᵢ, is a *child* of *r*.

3. **Parent.** The root, *r*, is the *parent* of each subordinate tree, *T*ᵢ.

4. **Path.** A *path* from node *n*₁ to *n*ₖ is defined as a sequence of nodes *n*₁, *n*₂, ..., *n*ₖ such that *n*ᵢ is the parent of *n*ᵢ₊₁ for 1 ≤ *i* ≤ *k*.

5. **Length.** The *length* of a path is the number of edges on the path. The length of the path is one less than the number of nodes on the path, namely *k* − 1.

6. **Depth.** The *depth* of a node *n*ᵢ is the length of the unique path from the root to *n*ᵢ. The root is at depth zero (0).

7. **Height.** The *height* of a node *n*ᵢ is the length of the longest path from *n*ᵢ to a leaf. All leaves are at height zero (0). The height of a tree is equal to the height of the root.

Examples from Figure 2.

1. **deanne** is a child of **alice**. **ilse** is a child of **edith**. **ilse** is the *grandchild* of **alice**.
2. **edith** is the *parent* of **julia**. **edith** is the *grandparent* of **paula**. **edith** is paula’s *grandmother*.
3. The path from **alice** to **qian** is **alice**, **edith**, **julia**, **qian**.
4. The length of the path from **alice** to **qian** is three (3).
5. **julia** is at depth two (2) because there are two edges on the path from **alice**, the root, to **julia**.
6. Julia is at height one (1) because the longest path to leaf is the path to Paula. The path to Paula has one edge. The height of the tree in Figure 2 is the height of Alice. The height of the tree is three because the longest path from Alice to a leaf contains three edges.

A binary tree is a tree in which no node can have more than two children.

Algorithms that operate on a binary tree are most efficient when the binary tree is completely filled with the possible exception of the bottom level.

Let $N$ be the number of the nodes in a completely filled binary tree. Let $h$ be the height of the tree.

$$2^h \leq N \leq 2^{h+1} - 1$$

For any tree that is entirely filled having the bottom level filled as well,

$$N = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

The number of comparisons to find a particular key is $h+1$, or $\left\lfloor \log_2 N \right\rfloor + 1$

The height of the tree $h = \left\lfloor \log_2 N \right\rfloor$. 
Binary search trees have an order property. Values stored in nodes to the left of node \( n \) are less than the value in \( n \), and values stored in nodes to the right of \( n \) are greater than the value in \( n \).

![Binary search tree](image)

**Figure 4.** Binary search tree

Every identifier to the left of lou is lexicographically less than lou and every identifier to the right of lou is lexicographically greater than lou. "Lexicographically" can be translated to "alphabetically."

Duplicates are prohibited. Every identifier in the binary tree is unique.

Node values are referred to as keys. Keys may have any type than can be compared using the comparison operators \(<, =, \text{ and } >\).

Binary trees are implemented using structures for nodes and separately allocated storage for identifiers as shown in Figure 5.

![Binary search tree diagram](image)

**Figure 5.** Binary search tree
class Tree {
    struct Node {
        Node* LNode;            // Left node (subtree)
        string Key;            // Key
        Node* RNode;           // Right node (subtree);
        Node(string k);
        void Print(ostream& o);
        void Print(ostream o,int depth);
    };
    Node* Root;              // Root of tree
    void Kill(Node* N);      // Remove all nodes starting with the root
    Node* Insert(Node* N,string Key);
    void PostOrder(Node* N,ostream& o);
    void PreOrder(Node* N,ostream& o);
    void InOrder(Node* N,ostream& o);
    void Graph(Node* N,int depth,ostream& o);
}
public:
    Tree();                  // Constructor
    ~Tree();                 // Destructor
    void Insert(string Key);  // Insert a key
    void PostOrder(ostream& o);  // Print the tree using a postorder traversal
    void PreOrder(ostream& o);  // Print the tree using a preorder traversal
    void InOrder(ostream& o);   // Print the tree using an inorder traversal
    void Graph(ostream& o);    // Print the tree using an inorder traversal
                                // where each node is indented according to its depth
};
Tree traversals include preorder, inorder, and postorder.

A preorder traversal of an expression tree is used to emit an expression in prefix form. Example: consider the expression in Figure 6 and the corresponding expression tree in Figure 7. Prefix notation for the expression is shown in Figure 8.

\[
(2 + 8) / 4 * (7 - 3)
\]

* / + 2 8 4 7 3

Figure 6. Expression

Figure 7. Expression tree for \((2+8)/4*(7-3)\)

void Tree::PreOrder(ostream& o) {PreOrder(Root,o);}
void Tree::PreOrder(Node* N,ostream& o)
1. Return if the value of parameter N is 0.
2. Print the identifier referenced from this node.
3. Visit the subordinate tree on the left.
4. Visit the subordinate tree on the right.

An inorder traversal prints the values of nodes in ascending order. An inorder traversal of the binary tree in Figure 4 produces the following list.

ann
dee
jan
lou
sue
zoe

By indenting the key according to the level of its node the following graphical presentation can be obtained.

   ann
     
     dee
       
       jan
     
     lou
       
       sue
         
         zoe
void Tree::Graph(ostream& o){Graph(Root,0,o); }
void Tree::Graph(Node* N,int depth,ostream& o)
1. Return if the value of parameter $N$ is 0.
2. Visit the subordinate tree on the left.
3. Print a new line
4. Indent according to the depth of the node.
5. Print the identifier referenced from this node.
6. Visit the subordinate tree on the right.

A postorder traversal of an expression tree is used to emit an expression in suffix notation.

A postorder traversal of the expression tree in Figure 7 produces the suffix form shown in Figure 9.

\[
2 8 + 4 / 7 3 - *
\]

Figure 9. Suffix form of \((2+8)/4*(7-3)\)

void Tree::PostOrder(ostream& o) { PostOrder(Root,o); }
void Tree::PostOrder(Node* N,ostream& o)
1. Return if the value of parameter $N$ is 0.
2. Visit the subordinate tree on the left.
3. Visit the subordinate tree on the right.
4. Print the identifier referenced from this node.

Node Constructor
Tree::Node::Node(string K):LNode(0),Key(K),RNode(0)\}

Tree Constructor
Tree::Tree():Root(0) \}

Tree Destructor
Tree::~Tree(){Kill(Root);}
Tree Insert

```cpp
void Tree::Insert(string Key) { Root=Insert(Root,Key); }

Tree::Node* Tree::Insert(Node* N,string Key)
{
    if (!N) return new Node(Key);
    if (Key==N->Key) return N;
    if (Key<N->Key)
        N->LNode=Insert(N->LNode,Key);
    else
        N->RNode=Insert(N->RNode,Key);
    return N;
}
```