

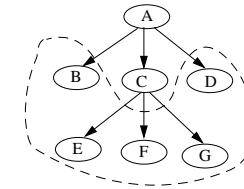
## Trees

Tree data structures, like the list, stack, and queue, data structures, are used all the time by software engineers. Tree data structures are evident in many everyday software applications. For example:

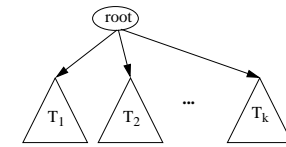
1. Language compilers (like C++) use tree data structures as part of the parsing phase of translation to machine code. For example, arithmetic expressions are represented using a *binary expression tree* that captures the precedence of arithmetic operators.
2. Search algorithms rely on tree structures to order data so that sub-linear search times are achievable. For example, a *binary search tree*, allows a search algorithm to find whether or not a key exists in  $O(\lg n)$  time.
3. Data compression algorithms require that encoded data be uniquely decodable. In Huffman coding, a code is constructed as a *binary coding tree* (also called a *binary trie*) that defines a unique *prefix code*, which guarantees unique decodability.
4. Operating systems provide a hierarchical tree structured file system to allow users to organize their files into directories (folders). The hierarchical tree structure guarantees a unique name for each directory or file.
5. Distributed file systems allow entire file systems to be attached as subtrees of other filesystem systems over a network. For example, the Network File System (NFS).
6. Distributed directories permit the distribution of information contained in subtrees for the purposes of administration and replication. For example, the Internet Domain Name System (DNS).
7. Databases rely on fast search of disk-based files using a key in order to retrieve a record. Many databases use a *b-tree* data structure that provides very fast keyed access to records stored on disk.

## General Trees

A general tree is a data structure that has a special **root node**, which has one or more **child nodes**. A tree is a *recursive* data structure, so each child node can itself be the root node for another set of child nodes.



root node, interior nodes, and leaf nodes



a root and its subtrees

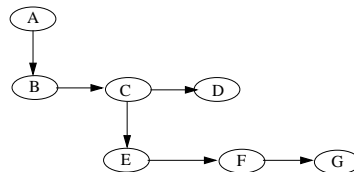
Nodes are connected by directed arcs called **edges**. A node is either the root node, an **interior** node, or a **leaf** node. The *leaves* of a tree are all the nodes with no outgoing edges. The root node has no **parent** node and one or more child nodes. An interior node has a single **parent** node, and one or more child nodes. A leaf node has a single parent node, and no child nodes. Since a node can only have a single parent, there is never more than one edge pointing at a node.

A tree with  $n$  nodes then has  $n-1$  edges, since every node but the root has one incoming edge.

The *depth* of a node is the length of the path from the root to the node. The root is at depth 0, and the depth of any node is 1 more than its parent node. The *height* of a tree is the length of the path from the node to the deepest leaf reachable from that node. The height of the root node is the height of the tree.

## General Tree Implementation

Since for any given node in a general tree, we cannot usually predict the number of children nodes, we cannot easily define a node structure/class that contains pointers to all the children nodes. Instead, we observe that a node has a first child, and some number of siblings. We can then define a child-sibling representation for a node, that will allow us to represent a tree structure programmatically.

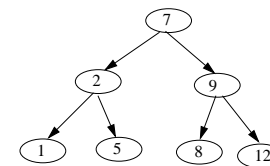


A child-sibling node may then be defined in C++ as follows:

```
template<class T> class TreeNode {
private:
    T elem;
    TreeNode *down, *right;
public:
    ...
    T* child() { return down; }
    T* sibling() { return right; }
};
```

## Binary Trees

A *binary tree* is a restriction on the general tree, such that any given node can have at most two child nodes: the *left child* and the *right child*. Some nodes may have one child, or no children.



To visit all nodes in the tree requires  $O(n)$  operations. Visiting all nodes in a tree is called traversing the tree.

A *binary search tree* is a binary tree that is organized in such a way that the time to find an entry in the tree is sublinear. Starting at the root of a binary search tree, we compare the element we want to find with the root, and if it is less, we go down the left branch, if the value is greater, we go down the right branch.

**Question:** given a binary search tree, what is the maximum number of nodes we need to visit to find a match or discover that the item does not exist?

**Question:** Can you represent a binary search tree using an array of sorted elements?

## Binary Tree Implementation

```
template<class T> class BinaryNode {
private:
    T elem;
    BinaryNode *left, *right;
public:
    BinaryNode() : left(NULL), right(NULL) {}
    ~BinaryNode() { delete left; delete right; }

    T* leftChild() { return left; }
    T* rightChild() { return right; }
    ...
    void printPreorder(ostream&) const;
    void printInorder(ostream&) const;
    void printPostorder(ostream&) const;
};
```

**Question:** Given the destructor shown above, what happens if you delete the root node of a binary tree?

## Tree Traversal Algorithms

There are three types of binary tree traversal.

1. Preorder traversal
2. Inorder traversal
3. Postorder traversal

A preorder traversal visits the root of a subtree, then the left and right subtrees recursively.

An inorder traversal visits the left subtree, the root of a subtree, and then the right subtree recursively.

A postorder traversal visits the left and right subtrees recursively, then the root node of the subtree.

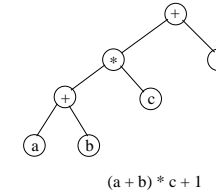
All of the traversal algorithms start at the root, and recursively go towards the leaves of the tree. Recursion is inherent in tree traversal, and all of the traversal algorithms are written as recursive procedures.

You should always check for a NULL left or right child pointer, since some nodes may have only one child node, and leaf nodes have no children nodes.

## A Binary Expression Tree

A binary expression tree captures the precedence of operators, so that expression evaluation is done correctly. The interior nodes contain binary operators and the leaf nodes contain variables or constant values that are to be evaluation.

A binary expression tree is evaluated from the left to right, from the leaf nodes up to the root. For example:



To compute the value of the expression, the tree must be traversed starting at the root. So, how does one traverse the tree such that the expression is evaluated correctly?

**Question:** How would you implement a simple calculator to evaluate expressions?

## Recursive Traversal Routines

```
template<class T>
void BinaryNode<T>::printPreorder(ostream& os) const
{
    os << elem << endl;
    if (left)
        left->printPreorder(os);
    if (right)
        right->printPreorder(os);
}

template<class T>
void BinaryNode<T>::printInorder(ostream& os) const
{
    if (left)
        left->printInorder(os);
    os << elem << endl;
    if (right)
        right->printInorder(os);
}

template<class T>
void BinaryNode<T>::printPostorder(ostream& os) const
{
    if (left)
        left->printPostorder(os);
    if (right)
        right->printPostorder(os);
    os << elem << endl;
}
```