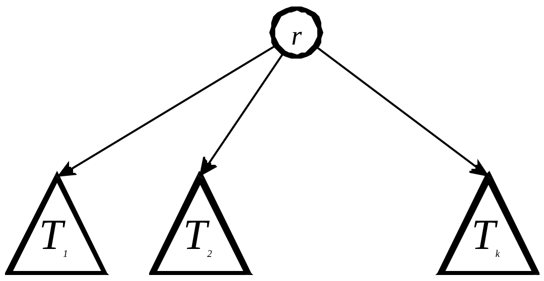

Figure 1. Generic tree

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 $1 \le i \le k$.
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.
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 a 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.

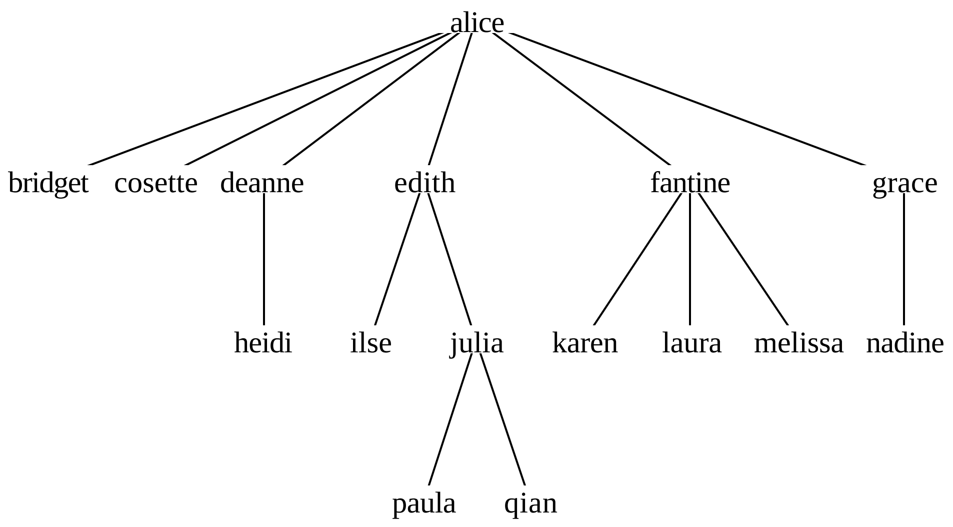

Figure 2. A tree

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.

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

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 height of the tree $h = \lfloor \log_2 N \rfloor$.

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

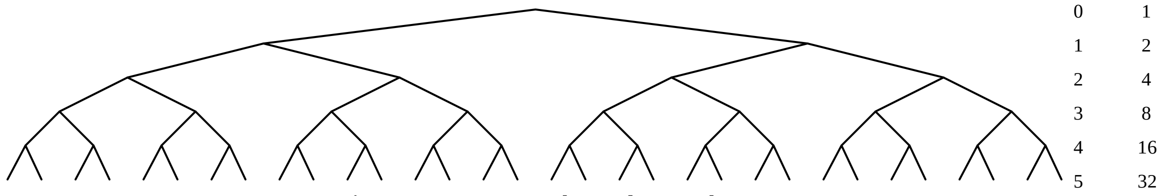

Figure 3. Counting the nodes in a binary tree

Binary search trees have an order property. Values stored in nodes to the *left* of node n_i are *less* than the value in n_i and values stored in nodes to the *right* of n_i are *greater* than the value in n_i.

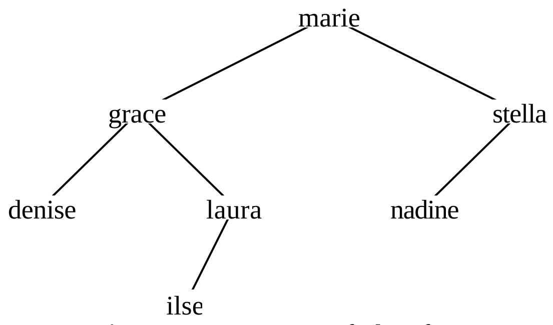

Figure 4. Binary tree of identifiers

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

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

Node values are refereed to as *keys*. Keys may have any type than can be compared using the comparison operators <, =, and >.

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

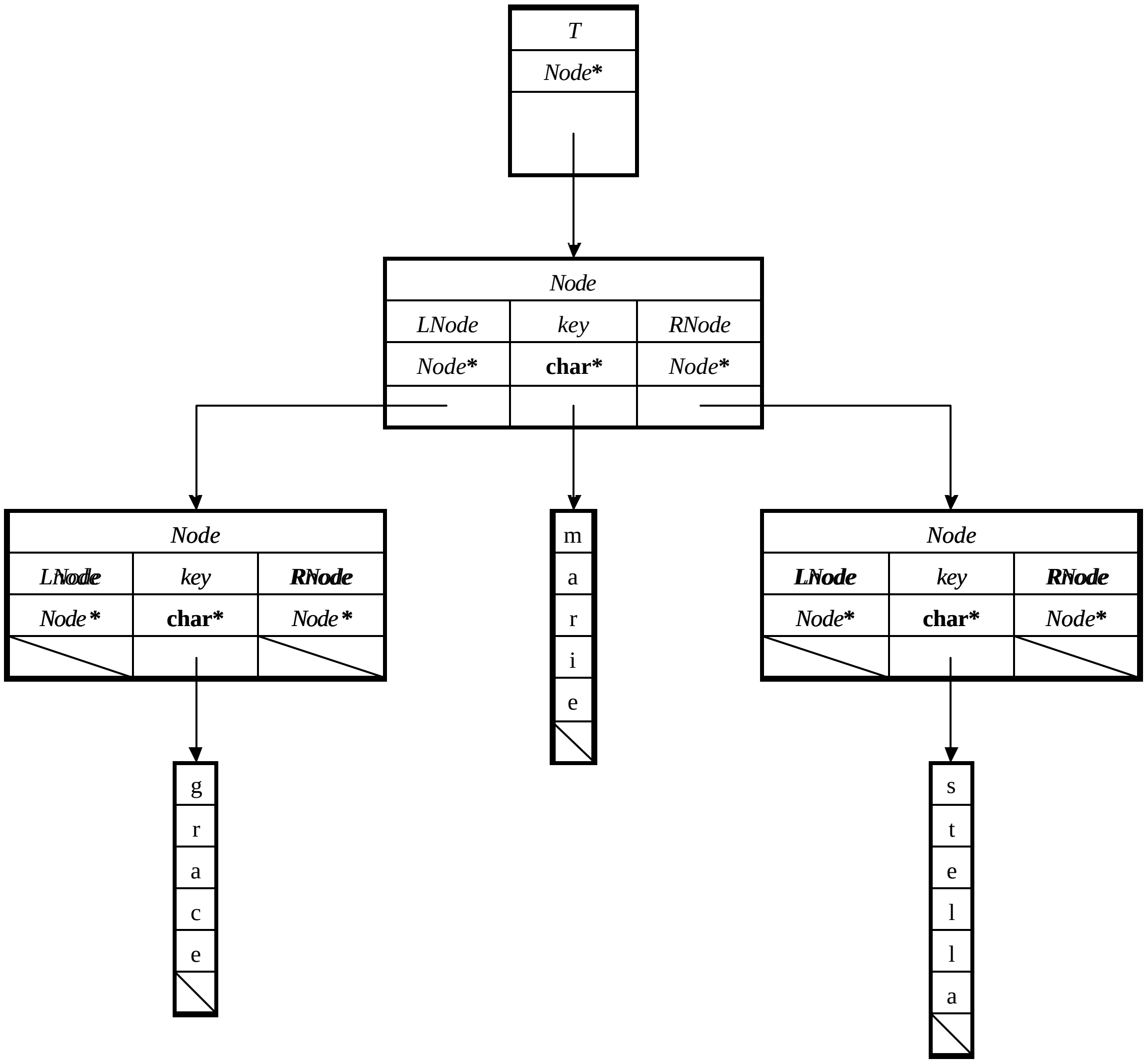

Figure 5. Anatomy of a binary tree**.**

```
class Tree {
      struct Node {
            Node* LNode;
            char* key;
            Node* RNode;
            …
      };
      Node* T;
      …
};
```

Figure 6. Binary tree class definition**.**

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)`

Figure 6. Expression

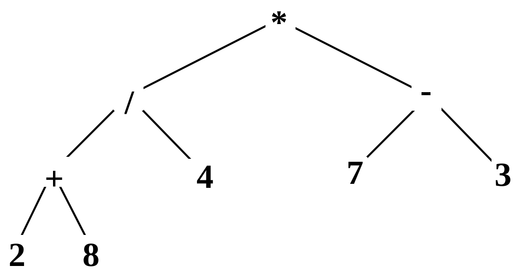

Figure 7. Expression tree for **(2+8)/4*(7-3)**

`* / + 2 8 4 - 7 3`

Figure 8. Prefix notation for **(2+8)/4*(7-3)**

void *Tree*::*PreOrder*(*Tree** *T*,*ostream*& *o*)
1. Return if the value of parameter *T* is `NULL`.
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.

desire
grace
ilse
laura
marie
nadine
stella

```
void Tree::InOrder(Tree* T,ostream& o)
1.  Return if the value of parameter T is NULL.
2.  Visit the subordinate tree on the left.
3.  Print the identifier referenced from this node.
4.  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(Tree* T,ostream& o)
1.  Return if the value of parameter t is NULL.
2.  Visit the subordinate tree on the left.
3.  Visit the subordinate tree on the right.
4.  Print the identifier referenced from this node.
```

Identifiers may be inserted into a binary tree using the following code.

```
#include <string>
#include <iostream>
using namespace std;
class Tree {
    struct Node {
        Node* LNode;
        char* key;
        Node* RNode;
        Node(char* k);
        ~Node();
    };
    Node* T;
    void Kill(Node* N);
    Node* Insert(Node* N,char* key);
public:
    Tree();
    ~Tree();
    void Insert(char* key);
};

Tree::Node::Node(char* k):LNode(0),RNode(0)
{   key:=new char[strlen(k)+1];
    strcpy(key,k);
}
Tree::Node::~Node() { if (key) delete[] key; }

void Tree::Kill(Node* N)
{   if  (n) {
        Kill(N->LNode);
        Kill(N->RNode);
        delete N;
    }
}
```

```
Tree::Tree():T(0) {}

Tree::~Tree() { Kill(T); }

Tree::Node* Tree::Insert(Node* N,char* key)
{     if (!N) return new Node(key);
      if (strcmp(key,N->key)==0) return N;
      if (strcmp(key,N->key)<0)
            N->LNode=Insert(N->LNode,key);
      else
            N->RNode=Insert(N->RNode,key);
      return N;
}

void Tree::Insert(char* key) {T=Insert(T,key); }

int main()
{     Tree T;
      T.Insert("marie");
      T.Insert("grace");
      T.Insert("stella");
      T.Insert("denise");
      T.Insert("laura");
      T.Insert("ilse");
      return 0;
}
```