```
template <class T>
void <Traversal_Method> (
                        TreeNode<T> *t,
                        void visit(T& item));
```

function parameter that accesses the data of the node

Recursive Tree Traversals

The tree traversal algorithms allow us to visit all the nodes in a tree.

The prefixes pre, in, and post indicate when the "visit" occurs at a node.

- **Inorder Traversal**
  - Traverse the left subtree
  - Visit the node
  - Traverse the right subtree

```
template <class T>
void Inorder(TreeNode<T> *t,
             void visit(T& item))
{
   if (t != NULL) {
      Inorder(t->Left(), visit);
      visit(t->data);
      Inorder(t->Right(), visit);
   }
}
```

- **Preorder Traversal**
- **Postorder Traversal**

Making Example Trees

```
void MakeCharTree(TreeNode<char>* &root, int n)
{
  // 9 TreeNode pointers; points to the 9 items in the tree
  TreeNode<char> *a, *b, *c, *d, *e, *f, *g, *h, *i;

  // parameter n specifies a tree in the range 0 - 2
  switch(n)
  {
    // nodes D and E leaf nodes ; A is root node
    case 0:
      d = GetTreeNode('D');
      e = GetTreeNode('E');
      b = GetTreeNode('B',(TreeNode<char> *)NULL,d);
      c = GetTreeNode('C',e, (TreeNode<char>*)NULL);
      a = GetTreeNode('A',b, c);
      root = a;
      break;
```



```
// nodes E, G, H, and I leaf nodes ; A is root node
    case 1:
      g = GetTreeNode('G');
      h = GetTreeNode('H');
      i = GetTreeNode('I');
      d = GetTreeNode('D');
      e = GetTreeNode('E',g,
                      (TreeNode<char>*)NULL);
      f = GetTreeNode('F',h, i);
      b = GetTreeNode('B',d, e);
      c = GetTreeNode('C',
                      (TreeNode<char> *)NULL,f);
      a = GetTreeNode('A',b, c);
      root = a;
      break;
```



```
// nodes F, G, H, and I leaf nodes ; A is root node
    case 2:
      g = GetTreeNode('G');
      h = GetTreeNode('H');
      i = GetTreeNode('I');
      d = GetTreeNode('D',
                      (TreeNode<char> *)NULL, g);
      e = GetTreeNode('E',h, i);
      f = GetTreeNode('F');
      b = GetTreeNode('B',d,
                      (TreeNode<char> *)NULL);
      c = GetTreeNode('C',e, f);
      a = GetTreeNode('A',b, c);
      root = a;
      break;
  }
}
```


Counting Leaf Nodes of a Binary Tree

```
template <class T>
void CountLeaf (TreeNode<T> *t, int& count)
{
  // use posorder descent
  if (t != NULL) {
    CountLeaf(t->Left(), count);  // descend left
    CountLeaf(t->Right(), count); // descend right

    // check if t is a leaf node (no descendants)
    // if so increment the variable count
    if (t->Left() == NULL && t->Right() == NULL)
      count++;
  }
}
```

Computing Depth of a Binary Tree

```
template <class T>
int Depth (TreeNode<T> *t)
{
   int depthLeft, depthRight, depthval;

   if (t == NULL)
      depthval = -1;
   else {
      depthLeft= Depth(t->Left());
      depthRight= Depth(t->Right());
      depthval = 1 +
(depthLeft>depthRight?depthLeft:depthRight);
   }
   return depthval;
}
```

The resulting depth of the node is 1 more than the maximum depth of its subtrees.

7

Tree Print

```
//spacing between levels
const int  indentBlock = 6;

//inserts num blanks on the current line
void IndentBlanks(int num)
{
    for (int i=0; i< num; i++)
          count << " ";
}

//print a tree sideways using an RNL tree scan
template <class T>
void PrintTree(TreeNode<T> *t, int level)
{
    if (t != NULL) {
          PrintTree(t->Right(), level+1);
          InsertBlanks(indentUnit * level);
          cout << t->data << endl;
          PrintTree(t->Left(), level+1);
    }
}
```

8

Copying a Tree

```
template <class T>
TreeNode<T> * CopyTree (TreeNode<T> *t)
{
    TreeNode<T> *newlptr, *newrptr, *newnode;

    if (t == NULL) return NULL;

    if (t->Left() != NULL)
          newlptr = CopyTree(t->Left());
    else
          newlptr = NULL;

    if (t->Right() != NULL)
          newrptr = CopyTree(t->Right());
    else
          newrptr = NULL;

    newnode = GetTreeNode(t->data, newlptr,newrptr);

    return newnode;
}
```

9

Deleting a Tree

```
template <class T>
void  DeleteTree (TreeNode<T> *t)
{
   if (t != NULL)
   {
      DeleteTree(t->Left());
      DeleteTree(t->Right());
      FreeTreeNode(t);
   }
}
```

10

Test: CopyTree and DeleteTree

```
#include <iostream.h>
#include <ctype.h>
#include <stdlib.h>

#include "treescan.h"
#include "treelib.h"
#include "treeprnt.h"

//used to lowercase char data values during
   postorder scan
void LowerCase(char &ch)
{
    ch = tolower(ch);
}
```

11

```
void main(void)
{
   // pointers for original and copied tree
   TreeNode<char> *root1, *root2;

   // create Tree_0 and print it
   MakeCharTree(root1, 0);
   PrintTree (root1, 0);

   // copy the tree so root is root2
   cout << endl << "Copy:" << endl;
   root2 = CopyTree(root1);

   // do postorder scan and then print tree.
   // <char> added due to a bug in Microsoft Visual C++
   Postorder<char> (root2, LowerCase);
   PrintTree (root2, 0);
}
```

12

```
/*
<Run of Program 11.2>

    C
        E
A
        D
    B

Copy:
    c
        e
a
        d
    b
*/
```

13

Breadth First Scan (Level Scan)

- visits all nodes (siblings) on the same level and then descend to the next level.
  - Rather than a recursive descent, we develop an iterative algorithm that uses a queue to hold the items.

```
template <class T>
void LevelScan(TreeNode<T> *t, void visit(T& item) {
   Queue<TreeNode<T> *> Q;
   TreeNode<T> *p;

    // initialize the queue by inserting the root
   Q.QInsert(t);

   // continue iterative process until queue is empty
   while( ! Q.QEmpty()) {
      // delete front node and execute the visit function
      p = Q.QDelete();
      visit(p->data);

      // if a left child exists, insert it in the queue
      if (p->Left() != NULL)
         Q.Qinsert(p->Left() );
      if (p->Right() != NULL)
         Q.Qinsert(p->Right() );
   }
}
```

14