National University of Computer & Emerging Sciences

Trees – Binary Search Trees

Deleting a Node

To delete a leaf node is easy -

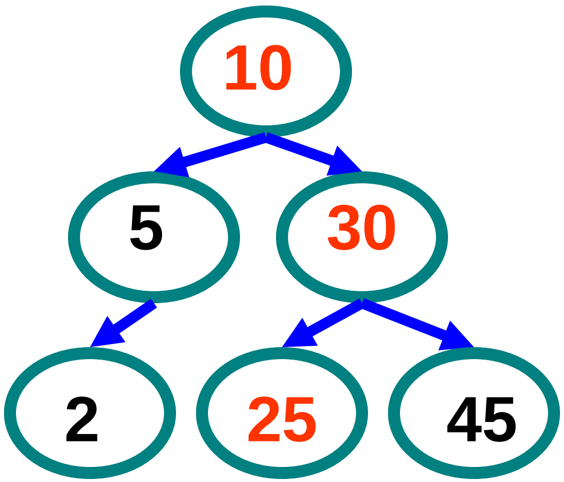
a) Find its parent

b) Set the child pointer that links to it to NULL

c) Free the node's memory

Example Deletion (Leaf)

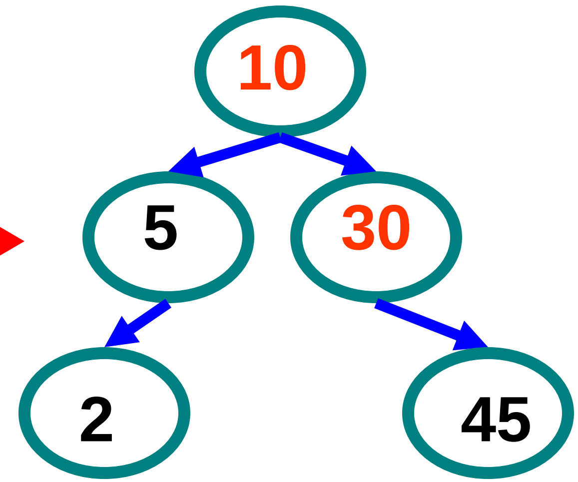
- Delete (25)



10 < 25, right

30 > 25, left

25 = 25, delete

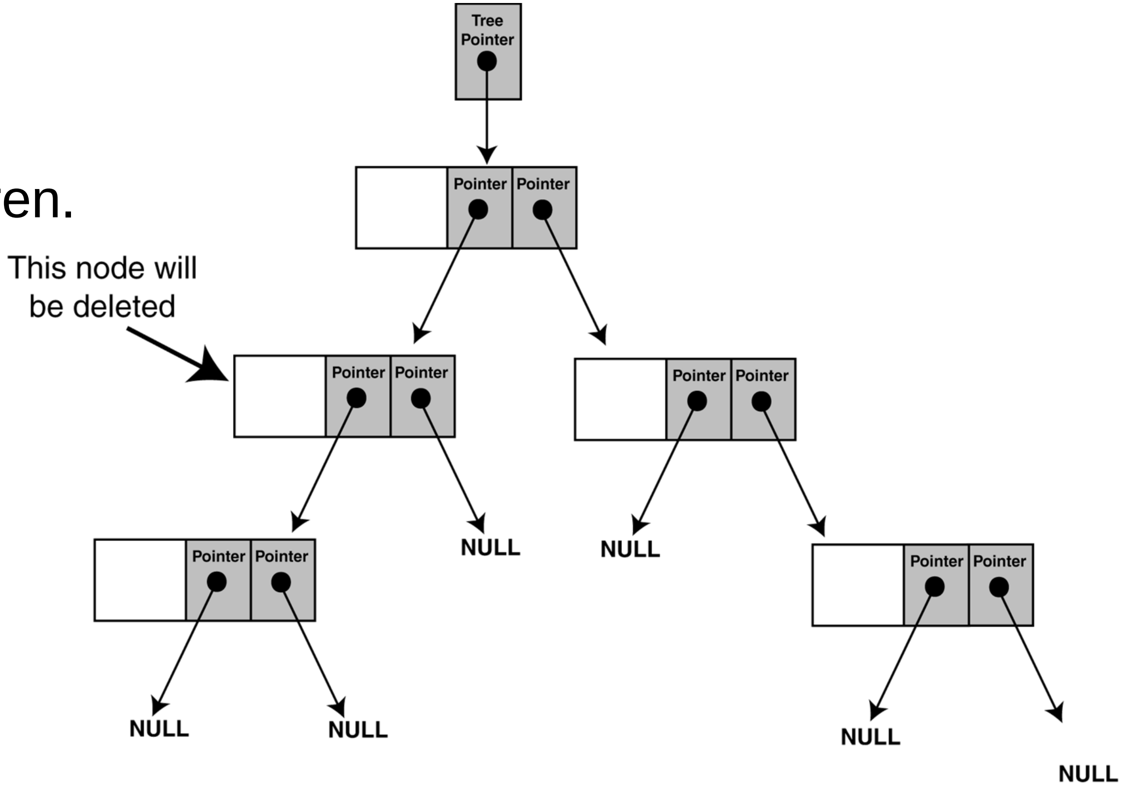


How to delete a node if it has child nodes?

We want to delete the node, but preserve the sub-trees that the node links to.

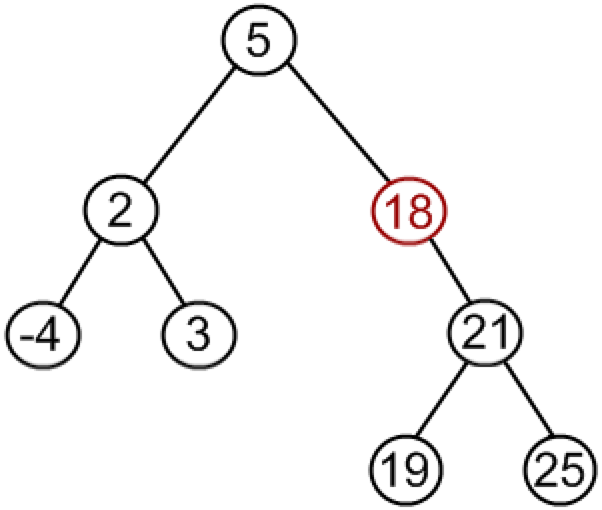
There are 2 possible situations to be faced when deleting a non leaf node -

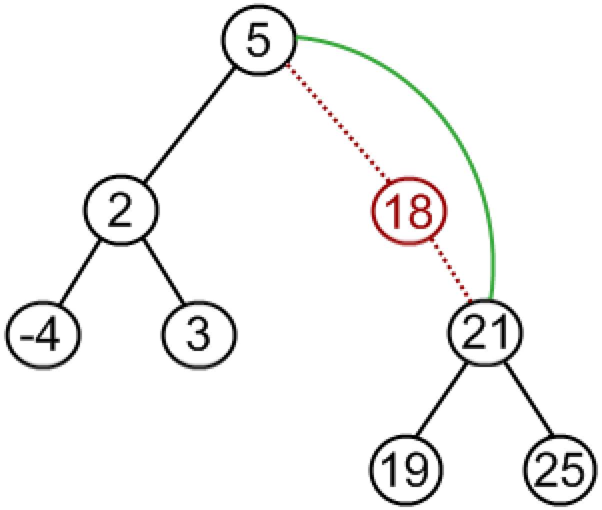
1) The node has one child.
2) The node has two children.

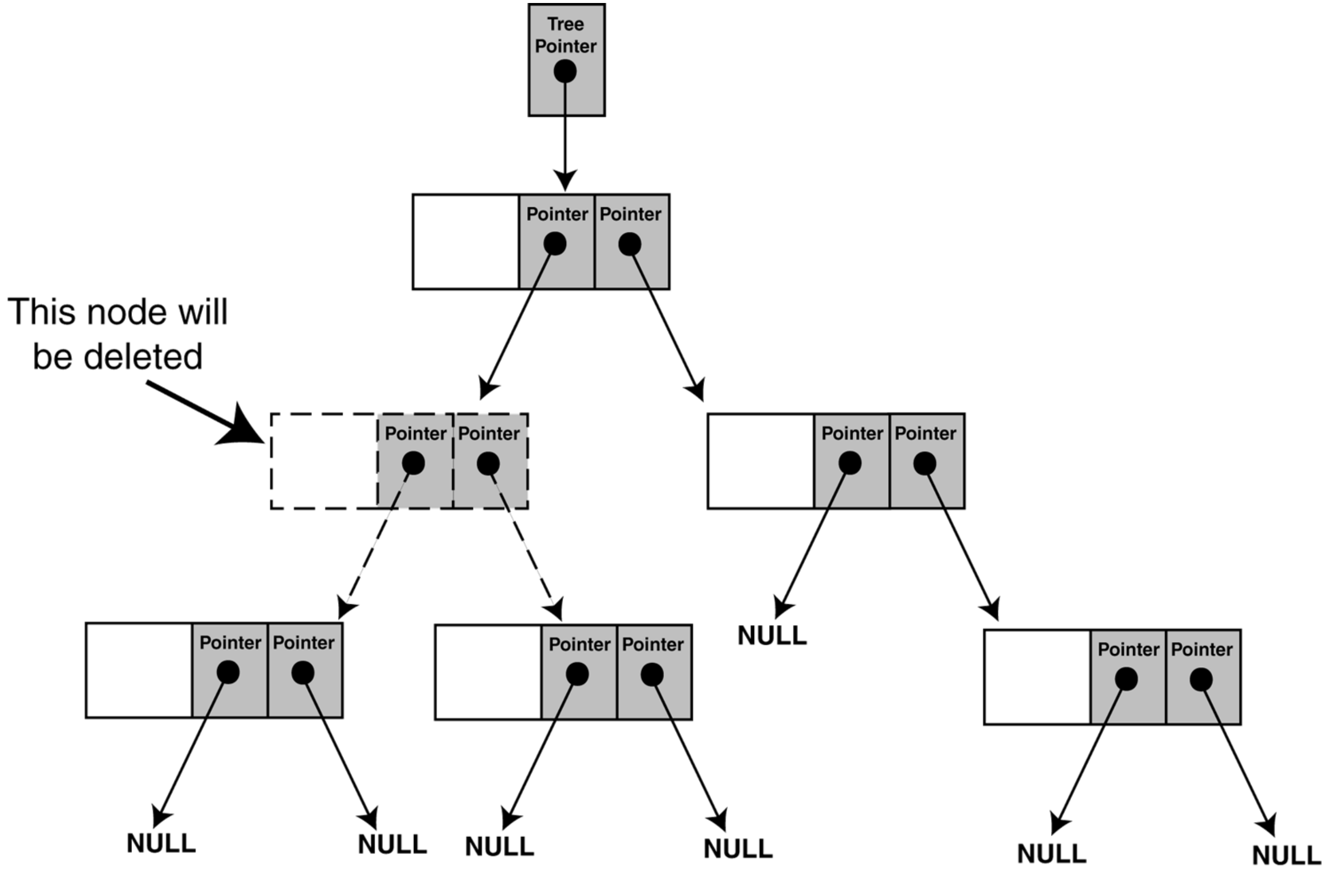


Example: Node to be removed has one child..

- Remove 18







The problem is not as easily solved if the node has two children.

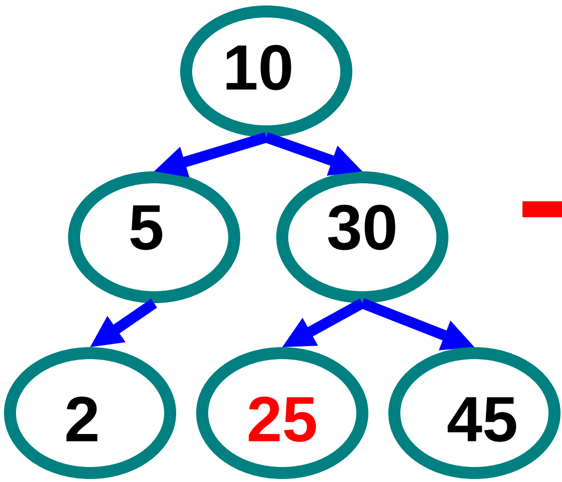
Binary Search Tree – a simpler method

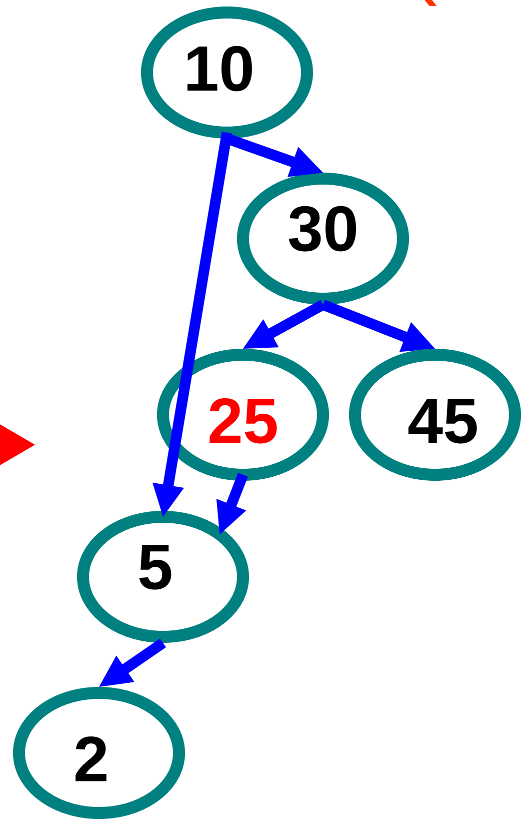
Find a position in the right subtree to attach the left subtree.

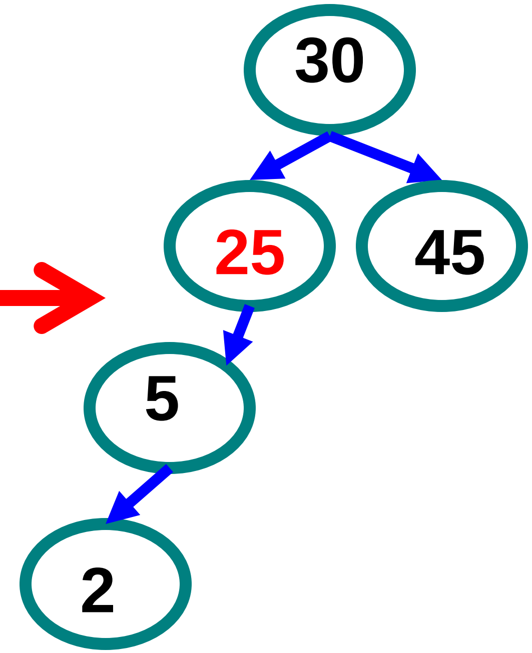
Attach the node's right subtree to the parent.

Example Deletion (Iterative)

- Delete (10)







Find the left-most node in right sub-tree

Attach the left sub-tree to that position

Attach node's <u>right subtree</u> to <u>parent, Delete</u>

Now the code – for the iterative version

But before that, some review

Pointers review

- Pointer to Pointer and reference to Pointer?

```
int g_One=1;

void func(int* pInt);

int main()
{
  int nvar=2;
  int* pvar=&nvar;
  func(pvar);
  std::cout<<*pvar<<std::endl;
  return 0;
}

void func(int* pInt)
{
  pInt=&g_One;
}
```

```
int g_One=1;

void func(int*& rpInt);

int main()
{
  int nvar=2;
  int* pvar=&nvar;
  func(pvar);
  std::cout<<*pvar<<std::endl;
  return 0;
}

void func(int*& rpInt)
{
rpInt=&g_One;
}
```

Binary Search Trees

Now the code - to delete a node from the IntBinaryTree, call the public member **remove**. The argument passed to the function is the value of the node you want to delete.

Note: The value may not exist in the tree so we need to find the node that contains the value to delete.

```
class IntBinaryTree
{
private:
        TreeNode *root;
        void destroySubTree(TreeNode *);
        void deleteNode(int, TreeNode *&);
        void makeDeletion(TreeNode *&);
        void displayInOrder(TreeNode *);
        void displayPreOrder(TreeNode *);
        void displayPostOrder(TreeNode *);
  public:
      IntBinaryTree() { root = NULL; } // Constructor
      ~IntBinaryTree() { destroySubTree(root); }
      void insertNode(int);
      bool searchNode(int);
      void remove(int);
      void showNodesInOrder(void)
          {       displayInOrder(root); }
      void showNodesPreOrder()
          {       displayPreOrder(root); }
      void showNodesPostOrder()
          {       displayPostOrder(root); }
};
```

```
void IntBinaryTree::remove(int num)
{
		deleteNode(num, root);
}
```

The remove member function calls the deleteNode member function. It passes the value of the node to delete, and the root pointer.

The deleteNode member function is shown below -

```
void IntBinaryTree::deleteNode(int num, TreeNode *&nodePtr)
{
        if (nodePtr == NULL)                    // node does not exist in the tree
          cout << num <<“ not found.\n";
        else if (num < nodePtr->value)
          deleteNode(num, nodePtr->left);       // find in left sub-tree
        else if (num > nodePtr->value)
          deleteNode(num, nodePtr->right);      // find in right sub-tree
        else
          makeDeletion(nodePtr);                // num == nodePtr->value i.e. node is found
}
```

Notice the declaration of the nodePtr parameter:

```
TreeNode *&nodePtr;
```

nodePtr is not simply a pointer to a TreeNode structure, but a *reference* to a pointer to a TreeNode structure. Any action performed on nodePtr is actually performed on the argument passed into nodePtr

The reason for doing this is explained shortly.

The *deleteNode* function uses an if/else statement.

```
if(num < nodePtr->value)
    deleteNode(num, nodePtr->left);
```

The above statement compares the parameter num with the value member of the node that nodePtr point to.

If num is less, the value being searched for will appear somewhere in the nodePtr's left subtree (if it appears at all).

So recall the deleteNode function recursively, with num as the first argument, and nodePtr->left as the second argument.

If num is not less than nodePtr->value, then the following else if statement is executed.

```
else if(num > nodePtr->value)
    deleteNode(num, nodePtr->right);
```

If num is greater than nodePtr->value, then the value being searched for will appear somewhere in nodePtr's right subtree (if it appears in the tree at all).

If num is equal to nodePtr->value, then neither of the if statements above will find a true condition.

So, nodePtr points to the node to be deleted, and the trailing else will be executed.

```
else
   makeDeletion(nodePtr);
```

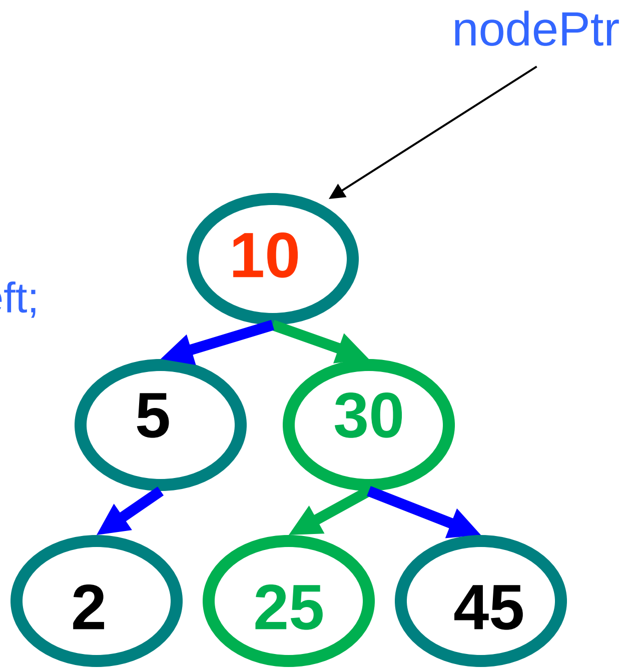
The makeDeletion function actually **deletes** the node from the tree and **reattaches** the deleted node's sub trees.

It must have access to the actual pointer in the tree to the node that is being deleted (not just a copy of the pointer).

This is why the *nodePtr* parameter in the deleteNode function is a reference. It must pass to makeDeletion, the actual pointer, to the node to be deleted.

```
void IntBinaryTree::makeDeletion(TreeNode *&nodePtr)
{
	TreeNode *tempNodePtr;	// Temporary pointer, used in
				// reattaching the left sub-tree

	if (nodePtr->right == NULL) // case for leaf and one (left) child
	{
		tempNodePtr = nodePtr;
		nodePtr = nodePtr->left; // Reattach the left child
		delete tempNodePtr;
	}
```

```
        else if (nodePtr->left == NULL) // case for one (right) child
        {
                tempNodePtr = nodePtr;
                nodePtr = nodePtr->right; // Reattach the right child
                delete tempNodePtr;
        }

        else  // case for two children.
        {
                // Move one node to the right.
                tempNodePtr = nodePtr->right;
                // Go to the extreme left node.
                while (tempNodePtr->left)
                        tempNodePtr = tempNodePtr->left;
                // Reattach the left subtree.
                tempNodePtr->left = nodePtr->left;
                tempNodePtr = nodePtr;
                // Reattach the right subtree.
                nodePtr = nodePtr->right;
                delete tempNodePtr;
        }
}
```



Program

```
// This program builds a binary tree with 5 nodes.
// The DeleteNode function is used to remove two
// of them.
#include <iostream.h>
#include "IntBinaryTree.h“

void main(void)
{
        IntBinaryTree tree;

        cout << "Inserting nodes.\n";
        tree.insertNode(5);
        tree.insertNode(8);
        tree.insertNode(3);
        tree.insertNode(12);
        tree.insertNode(9);

        cout << "Here are the values in the tree:\n";
        tree.showNodesInOrder();
```

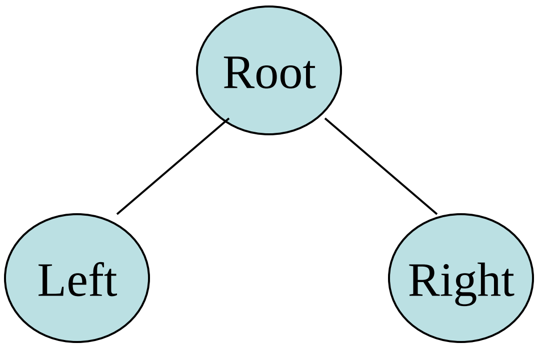
```
        cout << "Deleting 8...\n";
        tree.remove(8);
        cout << "Deleting 12...\n";
        tree.remove(12);

        cout << "Now, here are the nodes:\n";
        tree.showNodesInOrder();
}
```

Program Output

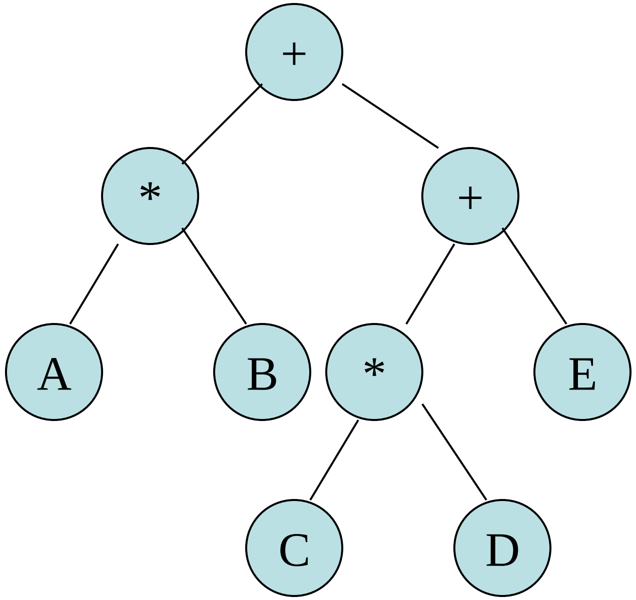
```
Inserting nodes.
Here are the values in the tree:
3
5
8
9
12
Deleting 8...
Deleting 12...
Now, here are the nodes:
3
5
9
```

Trees Traversal

- Inorder
  - (Left) Root (Right)
- Preorder
  - Root (Left) (Right)
- Postorder
  - (Left) (Right) Root



Trees Traversal



(A*B)+(C*D+E)

+*AB + *C D E

AB* C D * E + +

The IntBinaryTree class can *display* all the values in the tree using all 3 of these algorithms.

The algorithms are initiated by the following inline public member functions -

```
void showNodesInOrder(void)
        {       displayInOrder(root); }
void showNodesPreOrder()
        {       displayPreOrder(root); }
void showNodesPostOrder()
        {       displayPostOrder(root); }
```

Each of these public member functions calls a *recursive* private member function, and passes the root pointer as argument.

The code for these recursive functions is simple ☺ -

```
void IntBinaryTree::displayInOrder(TreeNode *nodePtr)
{
        if (nodePtr)
        {
                displayInOrder(nodePtr->left);
                cout << nodePtr->value << endl;
                displayInOrder(nodePtr->right);
        }
}


void IntBinaryTree::displayPreOrder(TreeNode *nodePtr)
{
        if (nodePtr)
        {
                cout << nodePtr->value << endl;
                displayPreOrder(nodePtr->left);
                displayPreOrder(nodePtr->right);
        }
}
```

```
void IntBinaryTree::displayPostOrder(TreeNode *nodePtr)
{
        if (nodePtr)
        {
                displayPostOrder(nodePtr->left);
                displayPostOrder(nodePtr->right);
                cout << nodePtr->value << endl;
        }
}
```

Program

```
// This program builds a binary tree with 5 nodes.
// The nodes are displayed with inorder, preorder,
// and postorder algorithms.
#include <iostream.h>
#include "IntBinaryTree.h“

void main(void)
{
        IntBinaryTree tree;

        cout << "Inserting nodes.\n";
        tree.insertNode(5);
        tree.insertNode(8);
        tree.insertNode(3);
        tree.insertNode(12);
        tree.insertNode(9);
```

```
        cout << "Inorder traversal:\n";
        tree.showNodesInOrder();
        cout << "\nPreorder traversal:\n";
        tree.showNodesPreOrder();
        cout << "\nPostorder traversal:\n";
        tree.showNodesPostOrder();
}
```

Program Output

Inserting nodes.
Inorder traversal:
3
5
8
9
12

```
// Move one node the right.
tempNodePtr = nodePtr->right;
// Go to the end left node.
while (tempNodePtr->left)
    tempNodePtr = tempNodePtr->left;

// Reattach the left subtree.
tempNodePtr->left = nodePtr->left;
tempNodePtr = nodePtr;

// Reattach the right subtree.
nodePtr = nodePtr->right;
delete tempNodePtr;
```

Preorder traversal:

5

3

8

12

9

Postorder traversal:

3

9

12

8

5