```
 //////////////////////////////
//                            //
//                            //
//      BinaryExpTree.h       //
//                            //
//                            //
//////////////////////////////

#ifndef BINARY_EXP_TREE_H
#define BINARY_EXP_TREE_H

#include <iostream>   //Included for STL cout

#define OPERATION 0
#define VALUE     1

class ExpTreeNode
{
public:
   ExpTreeNode(int data); // Constructor for a node holding data
   ExpTreeNode(char op); // Constructor for a node holding an operation
   ~ExpTreeNode();       // Destructor
   int getType();     // Returns whether the node is an operation or a value
   int getData();        // Returns the data held by the node
   char getOp();         // Returns the operation held by the node
   int getValue(); // Returns the value of subtrees of an operation

private:
   int type;        // Whether this node is an operation or number
   int data;        // The data in this node
   char op;         // The operation of this
   TreeNode *left;   // Pointer to the left subtree
   TreeNode *right;  // Pointer to the right subtree
}

class BinaryExpTree
{

        private:
                ExpTreeNode *root;  //Root node of the binary tree
                void insertNode(ExpTreeNode *nextNode, ExpTreeNode *insertNode);/*Insert node into the
tree*/
                void removeNode(int data, ExpTreeNode* parent);/*Removes node from tree*/
                ExpTreeNode* getLeftMost(ExpTreeNode* node);/*Returns leftmost node of a subtree*/
                void printInorder(ExpTreeNode* node); /*Output the inorder of a subtree*/
                void printPostorder(ExpTreeNode *node); /*Output the postorder of a subtree*/
                void printPreorder(ExpTreeNode *node); /*Output the preorder of a subtree*/
                int calculate(ExpTreeNode *node, int val); /*Calculates the value of the trees expression*/

        public:
                BinaryTree();//Constructor
                ~BinaryTree();//Destructor
                void insertData(int dataToInsert);//Insert data into the tree
        void insertData(char opToInsert);//Insert operation into the tree
                void removeData(int dataToRemove);//Delete a node from the tree
                void printInorder(); //Output the inorder of the tree
                void printPostorder(); //Output the postorder of the tree
                void printPreorder(); //Output the preorder of the tree
                int calculate(); //Calculates the value of the trees expression

};
#endif
```

```
///////////////////////////////
//                           //
//                           //
//      BinaryExpTree.cpp    //
//                           //
//                                    //
//////////////////////////////

#include “BinaryExpTree.h”

/** Constructor of ExpTreeNode class **/
ExpTreeNode::ExpTreeNode(int data)
{
	type = VALUE;
	this->data = data;
	left = NULL;
	right = NULL;
}

/** Constructor of ExpTreeNode class **/
ExpTreeNode::ExpTreeNode(char op)
{
		type = OPERATION;
	this->op = op;
	left = NULL;
	right = NULL;
}

/** Destructor of ExpTreeNode class **/
ExpTreeNode::~ExpTreeNode()
{
	left = NULL;
	right = NULL;
}

/** Returns whether the node is a value or an operation **/
int ExpTreeNode::getType()
{
		return type;
}

/** Returns data held by the node **/
int ExpTreeNode::getData()
{
		if(type == VALUE)
				return data;
		else
				return -1;//Error
}

/** Return operation held by the node **/
char ExpTreeNode::getOp()
{
		if(type == OPERATION)
				return op;
		else
				return "";//Error
}
```

```
/** Return value of a subtree when performed by designated operation **/
int ExpTreeNode::getValue()
{
        if(type == OPERATION)
        {
                int ret;
                int leftVal = left->data;
        int rightVal = right->data;
        switch (op)
             {
          case '+':  return leftVal + rightVal;
          case '-':  return leftVal - rightVal;
          case '*':  return leftVal * rightVal;
          case '/':  return leftVal / rightVal;
                  case '%':  return leftVal % rightVal;
        }
        }
        else
                return -1;//Error
}



/** Constructor of BinaryTree class **/
BinaryExpTree::ExpBinaryTree()
{
        root = NULL;
}

/** Destructor of BinaryTree class **/
BinaryExpTree::~BinaryExpTree()
{
        delete root;
}

/** Inserts data into the BinaryTree
  @param the data to be inserted
*/
void BinaryExpTree::insertData(int dataToInsert)
{
        if(root == NULL)
                root->data = new ExpNode(dataToInsert);;
        else
        {
                ExpTreeNode *newNode = new ExpTreeNode(dataToInsert);
                insertNode(root, newNode);
        }
}

/** Inserts operation into the BinaryTree
  @param the operation to be inserted
*/
```

```
void BinaryExpTree::insertData(char opToInsert)
{
        if(root == NULL)
                root->data = new ExpNode(opToInsert);;
        else
        {
                ExpTreeNode *newNode = new ExpTreeNode(opToInsert);
                insertNode(root, newNode);
        }
}

/** Inserts a node into the binary tree
  @param the node we are inserting a node into
  @param the node being inserted into the tree
*/
void BinaryExpTree::insertNode(ExpTreeNode *nextNode, ExpTreeNode *insertNode)
{
        if(insertNode->data > nextNode->data)
        {
                if(nextNode->right == NULL)
                        nextNode->right = insertNode;
                else
                        insertNode(nextNode->right, insertNode);
        }

        if(insertNode->data <= nextNode->data)
        {
                if(nextNode->left == NULL)
                        nextNode->left = insertNode;
                else
                        insertNode(nextNode->left, insertNode);
        }
}

/** Removes data from the tree **/
void BinaryExpTree::removeData(int dataToRemove)
{
        removeNode(dataToRemove, root);
}

/** Removes a node from the tree
  @param the data contained in the node being removed
  @param the tree or subtree we are removing a node from
*/
void BinaryExpTree::removeNode(int data, ExpTreeNode *parent)
{
        if(parent == NULL)
                return;
        if(parent->data == data)
        {
                if((parent->right == NULL || parent->right->right == NULL) && parent->left != NULL)
                        parent = left;
                if(parent->left = NULL && parent->right != NULL)
```

```
                        parent = right;
                if(parent->left == NULL && parent->right == NULL)
                        delete parent;
                parent = getLeftMost(parent->right);
                return;
        }
        if(parent->data <= data)
                removeNode(data, parent->left);
        if(parent->data > data)
                removeNode(data, parent->right);
}

/** Returns the leftmost node of a tree or subtree
  @param the tree or subtree to return the leftmost node from
  @return the leftmost node of a tree or subtree
*/
ExpTreeNode* BinaryExpTree::getLeftMost(TreeNode* node)
{
        if(node->left == NULL)
                return node;
        else
                return getLeftMost(node->left);
}

/** Prints the inorder transversal of the binary tree **/
void BinaryExpTree::printInorder()
{
        printInorder(root);
}

/** Prints the inorder transversal of a tree or subtree
  @param the tree or subtree we are printing the inorder of
*/
void BinaryExpTree::printInorder(ExpTreeNode *node)
{
  if (node != NULL)
  {
     printInorder(node->left);
     std::cout << node->item << " ";
     printInorder(node->right);
   }
}

/** Prints the postorder transversal of the binary tree **/
void BinaryExpTree::printPostorder()
{
        printPostorder(root);
}

/** Prints the postorder transversal of a tree or subtree
  @param the tree or subtree we are printing the postorder of
*/
void BinaryExpTree::printPostorder(ExpTreeNode *node)
```

```
{
  if (node != NULL)
  {
     printPostorder(node->left);
     printPostorder(node->right);
          std::cout << node->item << " ";
   }
}

/** Prints the preorder transversal of the binary tree **/
void BinaryExpTree::printPreorder()
{
          printPreorder(root);
}

/** Prints the preorder transversal of a tree or subtree
  @param the tree or subtree we are printing the preorder of
*/
void BinaryExpTree::printPreorder(ExpTreeNode *node)
{
  if (node != NULL)
  {
 std::cout << node->item << " ";
     printPreorder(node->left);
     printPreorder(node->right);
   }
}

/** Calculates the value of the expression of the binary tree **/
int BinaryExpTree::calculate()
{
          return calculate(root, 0);
}

/** Calculates the value of the expression of the binary tree
  @param the tree or subtree we are calculating
  @param the value we've calculated so far
*/
int BinaryExpTree::calculate(ExpTreeNode *node, int val)
{
  if (node->getType != OPERATION)
  {
     val = calculate(node->left, val);
     return calculate(node->right, val);
  }
  else
    val = node->getValue();

  return val;
}
```

```
///////////////////////////////
//                           //
//                           //
//          main.cpp         //
//                           //
//                           //
///////////////////////////////

#include <iostream> //Included for STL cout

#include “BinaryExpTree.h”

void Test1();
void Test2();
void Test3();

int main()
{
        Test1();
        Test2();
        Test3();

        return 0;
```

```
}

/** Insert nodes into a binary tree and print the transversals **/
void Test1()
{
        BinaryExpTree *binaryExpTree = new BinaryExpTree();

        binaryExpTree->insertData(3);
    binaryExpTree->insertData('+');
        binaryExpTree->insertData(1);
    binaryExpTree->insertData('*');
        binaryExpTree->insertData(5);
    binaryExpTree->insertData('-');
        binaryExpTree->insertData(3);

        std::cout << “Inorder of tree : “;
        binaryExpTree->printInorder();
        std::cout << std::endl;
        std::cout << “Preorder of tree : “;
        binaryExpTree->printPreorder();
        std::cout << std::endl;
        std::cout << “Postorder of tree : “;
        binaryExpTree->printPostorder();
        std::cout << std::endl;
        std::cout << “Value of the expression of the tree : “;
        sts::cout << binaryExpTree->calculate() << std::endl;
}

/** Insert nodes into a binary tree, delete a node, and print the transversals **/
void Test2()
{
        BinaryExpTree *binaryExpTree = new BinaryExpTree();

        binaryExpTree->insertData(5);
        binaryExpTree->insertData('+');
        binaryExpTree->insertData(4);
        binaryExpTree->insertData('-');
        binaryExpTree->insertData(7);
        binaryExpTree->insertData('+');
        binaryExpTree->insertData(12);

        binaryExpTree->deleteData(7);

        std::cout << “Inorder of tree : “;
        binaryExpTree->printInorder();
        std::cout << std::endl;
        std::cout << “Preorder of tree : “;
        binaryExpTree->printPreorder();
        std::cout << std::endl;
        std::cout << “Postorder of tree : “;
        binaryExpTree->printPostorder();
        std::cout << std::endl;
        std::cout << “Value of the expression of the tree : “;
        sts::cout << binaryExpTree->calculate() << std::endl;
}


/** Insert nodes into a binary tree, delete the root node, and print the transversals **/
void Test3()
{
        BinaryExpTree *binaryExpTree = new BinaryExpTree();

        binaryExpTree->insertData(10);
        binaryExpTree->insertData('+');
        binaryExpTree->insertData(13);
        binaryExpTree->insertData('*');
        binaryExpTree->insertData(4);
        binaryExpTree->insertData('-');
        binaryExpTree->insertData(11);

        binaryExpTree->deleteData(10);

        std::cout << “Inorder of tree : “;
        binaryTree->printInorder();
        std::cout << std::endl;
        std::cout << “Preorder of tree : “;
        binaryTree->printPreorder();
        std::cout << std::endl;
        std::cout << “Postorder of tree : “;
        binaryTree->printPostorder();
        std::cout << std::endl;
        std::cout << “Value of the expression of the tree : “;
        sts::cout << binaryExpTree->calculate() << std::endl;
}
```

Program Output:

```
Inorder of Tree : 1 + 1 * 5 - 3
Preorder of Tree : + 1 1 * 5 - 3
Postorder of Tree : 5 * 3 - 1 1 +
Value of the expression of the tree: 3
Inorder of Tree : 5 + 4 - + 12
Preorder of Tree : + 5 4 - + 12
Postorder of Tree : 12 + - 5 4 +
Value of the expression of the tree: -3
Inorder of Tree : + 13 * 4 - 11
Preorder of Tree : + * 13 - 4 11
Postorder of Tree : 5 4 6 8 15 13 11
Value of the expression of the tree: 41
```