```
#ifndef BINARY_TREE_H_
#define BINARY_TREE_H_
#include <iostream>
#include <cstdlib>

#define SUCCESS 1
#define FAILURE -1

class BinTreeNode{
public:
	BinTreeNode(void* data);
	BinTreeNode* getLeftNode() const;
	BinTreeNode* getRightNode() const;
	/*Create a left node with value pointed to by "data" pointer*/
	int setLeftNode(void* data);
	/*Create a right node with value pointed to by "data" pointer*/
	int setRightNode(void* data);
	/*Set left pointer of current node to NULL*/
	int setLeftToNull();
	/*Set right pointer of current node to NULL*/
	int setRightToNull();
	bool isLeafNode() const;
	bool isEndOfBranch() const;
	void* getData() const;
	~BinTreeNode();

private:
	void *data;
	BinTreeNode* left;
	BinTreeNode* right;
};

class BinTree{
public:
	BinTree();
	~BinTree();
	int getTreeSize() const;
	BinTreeNode* getRoot() const;
	/*insert a node to left of parent node*/
	int insertLeft(BinTreeNode* parent,void *data);
	/*insert a node to right of parent node*/
	int insertRight(BinTreeNode* parent,void* data);
	/*recursively remove left sub-tree of parent node*/
	int removeLeft(BinTreeNode* parent);
	/*recursively remove right sub-tree of parent node*/
	int removeRight(BinTreeNode* parent);
	/*use this to destroy data only if data is allocated on the heap*/
```

```
        int destroyData(BinTreeNode* data);
        /*Count the number of the leaves in a tree*/
        int countLeaves(void);
        /*Count the number of non-leaves in a tree*/
        int countNonLeaves(void);
        /*Get the height of a tree*/
        int getHeight(void);
        /*print tree elements in pre-order by using user-defined print function*/
        void printPreOrder(void (*print)(const void *data));
        /*print tree elements in-order by usin user-defined print function */
        void printInOrder(void (*print)(const void *data));
        /*print tree elements post-order by using user-defined print function*/
        void printPostOrder(void (*print)(const void *data));
        /*Remove all leaves of a tree*/
        void removeLeaves(void);

private:
        /*internal functions*/
        void doCountLeaves(BinTreeNode* node);
        int doGetHeight(BinTreeNode* node);
        void doPrintPreOrder(BinTreeNode* node,void (*print)(const void *data));
        void doPrintInOrder(BinTreeNode* node,void (*print)(const void *data));
        void doPrintPostOrder(BinTreeNode* node,void (*print)(const void *data));
        void doRemoveLeaves(BinTreeNode* node);
        /*root pointer of a tree*/
        BinTreeNode* root;
        /*total elements in a tree*/
        int size;
        /*no. of leaves in a tree*/
        int leafCount;
        /*height of a tree*/
        int height;
};

#endif
```

```
#include <iostream>
#include <cstdlib>
#include <cstdio>
#include "binTree.h"

using namespace std;


/*--------------Binary Tree Node definition-------------------*/

/*Tree Node Constructor*/
BinTreeNode::BinTreeNode(void *data):left(NULL),right(NULL),data(data){
}

BinTreeNode::~BinTreeNode(){
}

BinTreeNode* BinTreeNode::getLeftNode() const{
	return this->left;
}
BinTreeNode* BinTreeNode::getRightNode() const{
	return this->right;
}

void* BinTreeNode::getData() const{
	return data;
}


int BinTreeNode::setLeftNode(void* data) {
	this->left =  new BinTreeNode(data);
	return SUCCESS;
}


int BinTreeNode::setRightNode(void* data) {
	this->right =  new BinTreeNode(data);
	return SUCCESS;
}


int BinTreeNode::setLeftToNull() {
	this->left =  NULL;
	return SUCCESS;
}
```

```
int BinTreeNode::setRightToNull() {
	this->right =  NULL;
	return SUCCESS;
}

bool BinTreeNode::isLeafNode() const{
	if(this->left == NULL && this->right==NULL)
		return true;
	return false;
}


/*--------------Binary Tree definition-------------------*/

/*Binary Tree Constructor*/
BinTree::BinTree():size(0),root(NULL),leafCount(0),height(-1){
}

BinTree::~BinTree(){
}

int BinTree::getTreeSize() const{
	return this->size;
}

BinTreeNode* BinTree::getRoot() const{
	return this->root;
}


int BinTree::insertLeft(BinTreeNode* parent,void *data){
	/*If we are inserting a new root*/
	if(parent==NULL){
		if(this->size==0){
			root = new BinTreeNode(data);
			this->size++;
			return SUCCESS;
		}
		return FAILURE;
	}
	/*If left of parent is empty*/
	if(parent->getLeftNode()==NULL){
		/*Inserting node to left of parent*/
		if(parent->setLeftNode(data) == SUCCESS){
			this->size++;
			return SUCCESS;
		}
		}
		return FAILURE;

}

/*Insert a node to right of parent node*/
int BinTree::insertRight(BinTreeNode* parent,void* data){
		/*If we are inserting a new root*/
		if(parent==NULL){
			if(this->size==0){
				root = new BinTreeNode(data);
				this->size++;
				return SUCCESS;
			}
			return FAILURE;
		}
		/*If right of parent is empty*/
		if(parent->getRightNode()==NULL){
			/*Inserting node to right of parent*/
			if(parent->setRightNode(data) == SUCCESS){
				this->size++;
				return SUCCESS;
			}
		}
		return FAILURE;
}


int BinTree::removeLeft(BinTreeNode* parent){
		BinTreeNode *toRemove;
		/*No removal from emtpy tree*/
		if(this->size == 0)
			return FAILURE;
		/*If deleting from root*/
		if(parent == NULL)
			toRemove = this->root;
		else
			toRemove = parent->getLeftNode();
		/*recursively remove all nodes to left of parent*/
		if(toRemove != NULL){
			this->removeLeft(toRemove);
			this->removeRight(toRemove);
			//this->destroyData(toRemove);
		}
		delete(toRemove);
		this->size--;
		return SUCCESS;
```

```
}


int BinTree::removeRight(BinTreeNode* parent){
	BinTreeNode *toRemove;
	/*No removal from emtpy tree*/
	if(this->size == 0)
		return FAILURE;
	/*if deleting from root*/
	if(parent == NULL)
		toRemove = this->root;
	else
		toRemove = parent->getRightNode();
	/*recursively remove all nodes to right of parent*/
	if(toRemove != NULL){
		this->removeLeft(toRemove);
		this->removeRight(toRemove);
		//this->destroyData(toRemove);
	}
	delete(toRemove);
	this->size--;
	return SUCCESS;
}


int BinTree::destroyData(BinTreeNode* data){
	if(data){
		delete data;
		return SUCCESS;
	}
	else{
		cout<<"Nothing to delete\n";
		return SUCCESS;
	}
}


int BinTree::countLeaves() {
	leafCount = 0;
	doCountLeaves(root);
	return leafCount;
}

/*Recursively count the number of leaves*/
void BinTree::doCountLeaves(BinTreeNode* node){
	if(node == NULL)
		return;
```

```
        if(node->isLeafNode()){
                leafCount++;
        }
        doCountLeaves(node->getLeftNode());
        doCountLeaves(node->getRightNode());
}

/*Count number of non-leaves in the tree*/
int BinTree::countNonLeaves(){
        return(size - countLeaves());
}

/*Get the height of a tree*/
int BinTree::getHeight(){
        return (doGetHeight(root));
}

/*Recursively find height of tree. Tree with root alone has zero height*/
int BinTree::doGetHeight(BinTreeNode* node) {
        if(node==NULL)
                return -1;
        int l = doGetHeight(node->getLeftNode());
        int r = doGetHeight(node->getRightNode());
        /*Compute which side has a higher value*/
        height = 1 + std::max(l,r);
        return height;
}

/*-------------PRE-ORDER TRAVERSAL----------*/
void BinTree::printPreOrder(void (*print)(const void *data)){
        if(root==NULL)
                return;
        else
                doPrintPreOrder(this->root,print);
}

void BinTree::doPrintPreOrder(BinTreeNode* node,void (*print)(const void *data)){
        if(node==NULL)
                return;
        print(node);
        doPrintPreOrder(node->getLeftNode(),print);
        doPrintPreOrder(node->getRightNode(),print);

}

/*-------------IN-ORDER TRAVERSAL----------*/
void BinTree::printInOrder(void (*print)(const void *data)){
	if(root==NULL)
		return;
	else
		doPrintInOrder(this->root,print);
}

void BinTree::doPrintInOrder(BinTreeNode* node,void (*print)(const void *data)){
	if(node==NULL)
		return;
	doPrintInOrder(node->getLeftNode(),print);
	print(node);
	doPrintInOrder(node->getRightNode(),print);

}


/*-------------POST-ORDER TRAVERSAL----------*/
void BinTree::printPostOrder(void (*print)(const void *data)){
	if(root==NULL)
		return;
	else
		doPrintPostOrder(this->root,print);
}

void BinTree::doPrintPostOrder(BinTreeNode* node,void (*print)(const void *data)){
	if(node==NULL)
		return;
	doPrintPostOrder(node->getLeftNode(),print);
	doPrintPostOrder(node->getRightNode(),print);
	print(node);

}

/*Remove all leaves of a tree*/
void BinTree::removeLeaves(){
	if(root==NULL)
		return;
	else{
		doRemoveLeaves(root);
	}
}

/*recursively remove left and right leaves*/
void BinTree::doRemoveLeaves(BinTreeNode* node){
	if(node==NULL)
		return;
	/*If left(current node) is leaf, set left = NULL, delete left(current node)*/
```

```
		if ( (node->getLeftNode() != NULL)  ){
			if ( node->getLeftNode()->isLeafNode() == true ){
				printf("Removed %d\n",*((int*)node->getLeftNode()->getData()));
				node->setLeftToNull();
				delete node->getLeftNode();
				this->size--;
			}
		}
		/*If right(current node) is leaf, set right = NULL, delete right(current node)*/
		if ( (node->getRightNode() != NULL)  ){
			if ( node->getRightNode()->isLeafNode() == true ){
				printf("Removed %d\n",*((int*)node->getRightNode()->getData()));
				node->setRightToNull();
				delete node->getRightNode();
				this->size--;
			}
		}
		doRemoveLeaves(node->getLeftNode());
		doRemoveLeaves(node->getRightNode());
	}
```

```
#include "binTree.h"

void printNodeData(const void* node){
        BinTreeNode* elem = (BinTreeNode*) node;
        printf("%d\n",*(int*)elem->getData());
}

void BuildTreeOne(BinTree &t, int data[]){
        t.insertLeft(NULL,&data[0]);
        t.insertLeft(t.getRoot(),&data[1]);
        t.insertLeft((t.getRoot())->getLeftNode(),&data[2]);
        t.insertLeft((t.getRoot())->getLeftNode()->getLeftNode(),&data[3]);
        t.insertRight(t.getRoot(),&data[4]);
        t.insertLeft((t.getRoot())->getRightNode(),&data[5]);
        t.insertRight((t.getRoot())->getRightNode(),&data[6]);
        t.insertRight((t.getRoot())->getRightNode()->getRightNode(),&data[7]);
        t.insertRight((t.getRoot())->getRightNode()->getRightNode()-
>getRightNode(),&data[8]);
}

void BuildTreeTwo(BinTree &t, int data[]){
        t.insertLeft(NULL,&data[0]);
        t.insertLeft(t.getRoot(),&data[1]);
        t.insertLeft((t.getRoot())->getLeftNode(),&data[2]);
        t.insertLeft((t.getRoot())->getLeftNode()->getLeftNode(),&data[3]);
        t.insertRight((t.getRoot())->getLeftNode()->getLeftNode(),&data[4]);
        t.insertRight((t.getRoot())->getLeftNode(),&data[5]);
        t.insertRight(t.getRoot(),&data[6]);
        t.insertLeft((t.getRoot())->getRightNode(),&data[7]);
        t.insertRight((t.getRoot())->getRightNode(),&data[8]);
}


int main(){
        BinTree tree1, tree2;
        /*elements in pre-order*/
        int treeOneData[] = {1,2,4,7,3,5,6,8,9};
        int treeTwoData[] = {6,4,2,1,3,5,8,7,9};

        BuildTreeOne(tree1,treeOneData);
        BuildTreeTwo(tree2,treeTwoData);

        printf("no. of leaves in Tree 1 = %d\n",tree1.countLeaves());
        printf("no. of leaves in Tree 2 = %d\n",tree2.countLeaves());
        printf("\nno. of non-leaves in Tree 1 = %d\n",tree1.countNonLeaves());
        printf("no. of non-leaves in Tree 2 = %d\n",tree2.countNonLeaves());
        printf("\nHeight of Tree 1 = %d\n",tree1.getHeight());
```

```
    printf("Height of Tree 2 = %d\n",tree2.getHeight());

    printf("\n\nPrinting Tree1 in Pre-Order \n");
    tree1.printPreOrder(printNodeData);
    printf("Printing Tree1 in In-Order \n");
    tree1.printInOrder(printNodeData);
    printf("Printing Tree1 in Post-Order \n");
    tree1.printPostOrder(printNodeData);

    printf("\n\nPrinting Tree2 in Pre-Order \n");
    tree2.printPreOrder(printNodeData);
    printf("Printing Tree2 in In-Order \n");
    tree2.printInOrder(printNodeData);
    printf("Printing Tree2 in Post-Order \n");
    tree2.printPostOrder(printNodeData);

    printf("\nRemoving leaves from tree1\n");
    tree1.removeLeaves();
    printf("Printing Tree1 in Pre-Order after removal\n");
    tree1.printPreOrder(printNodeData);
    printf("Printing Tree1 in In-Order after removal \n");
    tree1.printInOrder(printNodeData);
    printf("Printing Tree1 in Post-Order after removal \n");
    tree1.printPostOrder(printNodeData);

    printf("\nRemoving leaves from tree2\n");
    tree2.removeLeaves();
    printf("Printing Tree2 in Pre-Order after removal \n");
    tree2.printPreOrder(printNodeData);
    printf("Printing Tree2 in In-Order after removal \n");
    tree2.printInOrder(printNodeData);
    printf("Printing Tree2 in Post-Order after removal \n");
    tree2.printPostOrder(printNodeData);

    return 0;

}
```

OUTPUT

```
Ram (master *) BinaryTree $ ./hw6
no. of leaves in Tree 1 = 3
no. of leaves in Tree 2 = 5

no. of non-leaves in Tree 1 = 6
no. of non-leaves in Tree 2 = 4

Height of Tree 1 = 4
Height of Tree 2 = 3


Printing Tree1 in Pre-Order
1
2
4
7
3
5
6
8
9
Printing Tree1 in In-Order
7
4
2
1
5
3
6
8
9
Printing Tree1 in Post-Order
7
4
2
5
9
8
6
3
1


Printing Tree2 in Pre-Order
6
4
2
1
3
5
8
```

```
7
9
Printing Tree2 in In-Order
1
2
3
4
5
6
7
8
9
Printing Tree2 in Post-Order
1
3
2
5
4
7
9
8
6

Removing leaves from tree1
Removed 7
Removed 5
Removed 9
Printing Tree1 in Pre-Order after removal
1
2
4
3
6
8
Printing Tree1 in In-Order after removal
4
2
1
3
6
8
Printing Tree1 in Post-Order after removal
4
2
8
6
3
1

Removing leaves from tree2
Removed 5
Removed 1
Removed 3
Removed 7
Removed 9
Printing Tree2 in Pre-Order after removal
6
4
2
8
Printing Tree2 in In-Order after removal
2
4
6
8
Printing Tree2 in Post-Order after removal
2
4
8
6
Ram (master *) BinaryTree $
```