```
// Lina Kang
// CS1D MW 2:30 - 5:00 PM
// 09/23/2020
// This program explores the concept of binary tree and
// tests out its abilities through inserting items,
// in-order, pre-order, post-order, breadth-first traversals
// and hierarchical relationships

// -------------- OUTPUT -----------------
/*
 Lina Kang
 CS1D MW 2:30 - 5:00 PM
 09/23/2020
 This program explores the concept of binary tree and
 tests out its abilities through inserting items,
 in-order, pre-order, post-order, breadth-first traversals
 and hierarchical relationships


**In-Order Traversal**

 5 12 13 18 19 24 25 29 33 44 49 55 59 77 89 109 118 288 1001

**Pre-Order Traversal**

 89 59 19 13 5 12 18 25 24 33 29 44 55 49 77 288 109 118 1001

**Post-Order Traversal**

 12 5 18 13 24 29 49 55 44 33 25 19 77 59 118 109 1001 288 89

**Breadth-First Traversal**

 89 59 288 19 77 109 1001 13 25 118 5 18 24 33 12 29 44 55 49

**Print By Level**

Level 0: 89
Level 1: 59 288
Level 2: 19 77 109 1001
Level 3: 13 25 118
Level 4: 5 18 24 33
Level 5: 12 29 44
Level 6: 55
Level 7: 49

**Print Relationships of Nodes**

Node: 5
 - Parent: 13
 - Children: 12

Node: 12
 - Parent: 5

Node: 13
 - Parent: 19
```

```
 - Children: 5 18

Node: 18
 - Parent: 13

Node: 19
 - Parent: 59
 - Children: 13 25

Node: 24
 - Parent: 25

Node: 25
 - Parent: 19
 - Children: 24 33

Node: 29
 - Parent: 33

Node: 33
 - Parent: 25
 - Children: 29 44

Node: 44
 - Parent: 33
 - Children: 55

Node: 49
 - Parent: 55

Node: 55
 - Parent: 44
 - Children: 49

Node: 59
 - Parent: 89
 - Children: 19 77

Node: 77
 - Parent: 59

Node: 89
 - Children: 59 288

Node: 109
 - Parent: 288
 - Children: 118

Node: 118
 - Parent: 109

Node: 288
 - Parent: 89
 - Children: 109 1001

Node: 1001
 - Parent: 288
```

```cpp
*/
// -------------- END OUTPUT -----------------

#include "binaryTree.h"

int main()
{
    // Input & Initialization
    NodeBinaryTree binaryTree;

    binaryTree.insert(89);

    Node * root = binaryTree.getroot();

    binaryTree.insert(59, root);
    binaryTree.insert(288, root);
    binaryTree.insert(19, root);
    binaryTree.insert(13, root);
    binaryTree.insert(5, root);
    binaryTree.insert(109, root);
    binaryTree.insert(12, root);
    binaryTree.insert(118, root);
    binaryTree.insert(25, root);
    binaryTree.insert(33, root);
    binaryTree.insert(1001, root);
    binaryTree.insert(18, root);
    binaryTree.insert(44, root);
    binaryTree.insert(77, root);
    binaryTree.insert(55, root);
    binaryTree.insert(24, root);
    binaryTree.insert(49, root);
    binaryTree.insert(29, root);



    // Output & Processing
    cout << " Lina Kang\n"
            " CS1D MW 2:30 - 5:00 PM\n"
            " 09/23/2020\n"
            " This program explores the concept of binary tree and\n"
            " tests out its abilities through inserting items,\n"
            " in-order, pre-order, post-order, breadth-first traversals\n"
            " and hierarchical relationships \n\n";

    cout << "\n**In-Order Traversal**\n\n";

    binaryTree.in_order(root);

    cout << "\n\n**Pre-Order Traversal**\n\n";

    binaryTree.pre_order(root);

    cout << "\n\n**Post-Order Traversal**\n\n";

    binaryTree.post_order(root);

    cout << "\n\n**Breadth-First Traversal**\n\n";
```

```
    
    binaryTree.breadth_first(root);
    
    cout << "\n\n**Print By Level**\n\n";
    
    binaryTree.printByLevel(root);
    
    cout << "\n**Print Relationships of Nodes**\n\n";
    
    binaryTree.printRelation(root);
}

#ifndef BINARYTREE_H_
#define BINARYTREE_H_

#include <vector>
#include <queue>
#include <iostream>

using namespace std;

struct Node
{
    int value;
    int level;
    Node * left;
    Node * right;
    Node * parent;
    Node() : value(0), level(0), left(NULL), right(NULL), parent(NULL) { }
};

//linked-list implementation
class NodeBinaryTree
{
public:
    NodeBinaryTree();
    int getsize() const;
    bool isempty() const;
    Node * getroot() const;
    
    void insert(int);
    void insert(int, Node*);
    
    bool isExternal(Node * v) const { return v->left == NULL && v->right == NULL; }
    bool isInternal(Node * v) const { return v->left != NULL || v->right != NULL; }
    
    void in_order(Node*) const;
    void post_order(Node*) const;
    void pre_order(Node*) const;
    void breadth_first(Node*) const;
    
    void printByLevel(Node *) const;
    void printRelation(Node *) const;
    
private:
    Node *root;
    Node *current;
```

```
    int size;
};
NodeBinaryTree::NodeBinaryTree()
{
    root = NULL;
    current = NULL;
    size = 0;
}
int NodeBinaryTree::getsize() const
{
    return size;
}
bool NodeBinaryTree::isempty() const
{
    return root == NULL ? true : false;
}
Node * NodeBinaryTree::getroot() const
{
    return root;
}
// insert function for ONLY root
void NodeBinaryTree::insert(int item)
{
  if(root!=NULL)
    insert(item, root);
  else
  {
      Node * newNode = new Node;
      root = newNode;
      root->value=item;
      root->left=NULL;
      root->right=NULL;
      size++;
  }
}
// insert function for descendants of root (recursive)
void NodeBinaryTree::insert(int item, Node * node)
{
    // compare the item to current node
    if(item < node->value)
    {
        //recursively repeat until an empty spot is found to insert
        if(node->left != NULL)
            insert(item, node->left);
        else
        {
            Node * newNode = new Node;

            node->left=newNode;

            newNode->value = item;
            newNode->left = NULL;
            newNode->right = NULL;
            newNode->parent = node;

            size++;
```

```
        }
    }
    else if(item >= node->value)
    {
        //recursively repeat until an empty spot is found to insert
        if(node->right != NULL)
            insert(item, node->right);
        else
        {
            Node * newNode = new Node;

            node->right = newNode;

            newNode->value = item;
            newNode->left = NULL;
            newNode->right = NULL;
            newNode->parent = node;

            size++;
        }
    }
}

// Prints the binary tree from leftmost node to rightmost node
// (the output is from smallest to largest in increasing order)
void NodeBinaryTree::in_order(Node * p) const
{
    if(p->left != NULL)
        in_order(p->left);
    cout << " " << p->value;
    if(p->right != NULL)
        in_order(p->right);
}

// Prints the binary tree where nodes are visited after its descendants
void NodeBinaryTree::post_order(Node * p) const
{
    if(p->left != NULL)
        post_order(p->left);
    if(p->right != NULL)
        post_order(p->right);
    cout << " " << p->value;
}

// Prints the binary tree where nodes are visited before its descendants
void NodeBinaryTree::pre_order(Node * p) const
{
    cout << " " << p->value;
    if(p->left != NULL)
        pre_order(p->left);
    if(p->right != NULL)
        pre_order(p->right);
}

// Prints the binary tree at a top-down approach visiting nodes by level
void NodeBinaryTree::breadth_first(Node * p) const
{
```

```
    Node * current;

    queue<Node *> que;                          // utilize a queue to add nodes
    que.push(p);                                // and add the descendants

    while(que.size() > 0)
    {
        // get first value from queue, print that value, pop the value
        current = que.front();
        que.pop();
        cout << " "<< current->value;

        // enqueue descendants if they exist
        if(current->left != NULL)
            que.push(current->left);
        if(current->right != NULL)
            que.push(current->right);
    }
}

// Prints the binary trees divided by levels and nodes within those levels
// (similar algorithm from breadth_first traversal)
void NodeBinaryTree::printByLevel(Node * p) const
{
    Node * current;

    queue<Node *> que;
    que.push(p);

    int level = 0;
    cout << "Level " << level << ": ";

    while(que.size() > 0)
    {
        current = que.front();
        que.pop();
        cout << current->value << " ";

        if(current->left != NULL)
        {
            // designate a level to each node
            // (since current->left is a child of current, their level is
            //  1 + current's level)
            current->left->level = current->level + 1;
            que.push(current->left);

        }
        if(current->right != NULL)
        {
            current->right->level = current->level + 1;
            que.push(current->right);
        }

        // to prevent the next if condition from bugging out when size == 0
        if(que.size() == 0)
            break;
        // if the next node in queue is starting at next level,
```

```cpp
        // print a new line and print "Level"
        else if(current->level < que.front()->level )
        {
            level++;
            cout << endl << "Level " << level << ": ";
        }

    }
    cout << endl;
}
// Print the relationship of the nodes
// (same algorithm from in-order traversal)
void NodeBinaryTree::printRelation(Node * p) const
{
    if(p->left != NULL)
        printRelation(p->left);

    // -------- Output ----------
    cout << "Node: "  << p->value;

    if(p != root)
        cout << "\n - Parent: "   << p->parent->value;
    if(!isExternal(p))
    {
        cout << "\n - Children: ";
        if(p->left != NULL)
            cout << p->left->value << " ";
        if(p->right != NULL)
            cout << p->right->value;
    }
    cout << endl << endl;
    // ------ End Output --------

    if(p->right != NULL)
        printRelation(p->right);
}

#endif /* BINARYTREE_H_ */


```