```cpp
#include<iostream>
#include<stack>
#include<queue>
using namespace std;

struct Node
{
    int value;
    Node* left;
    Node* right;
    bool turn;
    Node(int value)
    {
        this->value = value;
        left = nullptr;
        right = nullptr;
        turn = false;
    }
};
////////////////////////////////////////////////////////////////




class Iterator
{
public:
    virtual bool hasNext() = 0;
    virtual int Next() = 0;
};
////////////////////////////////////////////////////////////////
```

```cpp
// Pre-Order Iterator
class TreePreOrderIterator: public Iterator
{
    stack<Node*> Stack;
public:
    TreePreOrderIterator(Node * root)
    {
        if(root!=nullptr)
        {
            Stack.push(root);
        }
    }

    bool hasNext()
    {
        return (!Stack.empty());
    }

    int Next()
    {
        Node* current = Stack.top(); // get current node from top of stack
        Stack.pop();

        if(current->right) // push right child on stack if any
        {
            Stack.push(current->right);
        }

        if(current->left) // push left child on stack if any
        {
            Stack.push(current->left);
        }

        return current->value; // return current node's value
    }
};
///////////////////////////////////////////////////////////////
```

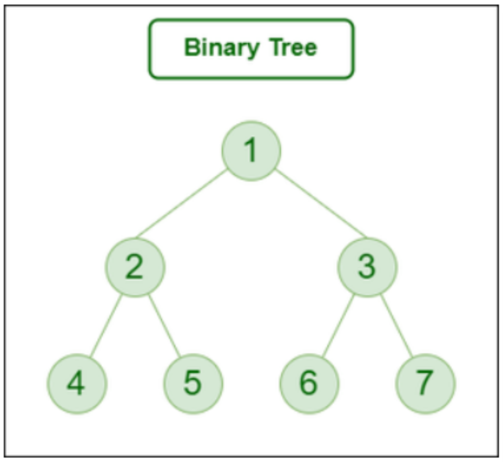
```cpp
// Post-Order Iterator
class TreePostOrderIterator : public Iterator
{
      stack<Node*> Stack;
public:
      TreePostOrderIterator(Node * root)
      {
            if(root!=nullptr)
            {
                  root->turn = false; // make the turn initially false
                  Stack.push(root);
            }
      }

      bool hasNext()
      {
            return (!Stack.empty());
      }

      int Next()
      {
            while(!Stack.empty())
            {
                  Node* current = Stack.top(); // get current node from top of stack
                  Stack.pop();

                  if(current->turn) // return value of the current node if its turn is true
                  {
                        return current->value;
                  }
                  else
                  {
                        current->turn = true; // make the turn of current node true if not true
                        Stack.push(current); // push current node back on the stack

                        if(current->right) // push right child on stack if any
                        {
                              current->right->turn = false; // make the turn false
                              Stack.push(current->right);
                        }

                        if(current->left) // push left child on stack if any
                        {
                              current->left->turn = false; // make the turn false
                              Stack.push(current->left);
                        }
                  }
            }
      }
};
////////////////////////////////////////////////////////////
```

```cpp
// In-Order Iterator
class TreeInOrderIterator: public Iterator
{
    stack<Node*> Stack;
    Node* current;
public:
    TreeInOrderIterator(Node * root)
    {
        current = root;
    }

    bool hasNext()
    {
        return (current != nullptr || !Stack.empty());
    }

    int Next()
    {
        while(current || !Stack.empty())
        {
            if(current) // process current node's left sub-tree first
            {
                Stack.push(current);
                current = current->left;
            }
            else // process a node and its right sub-tree
            {
                current = Stack.top(); // get current node from top of stack
                Stack.pop();

                int value = current->value; // save value of the current node

                current = current->right; // process current node's right sub-tree

                return value; // return current node's value
            }
        }
    }
};
///////////////////////////////////////////////////////////////
```

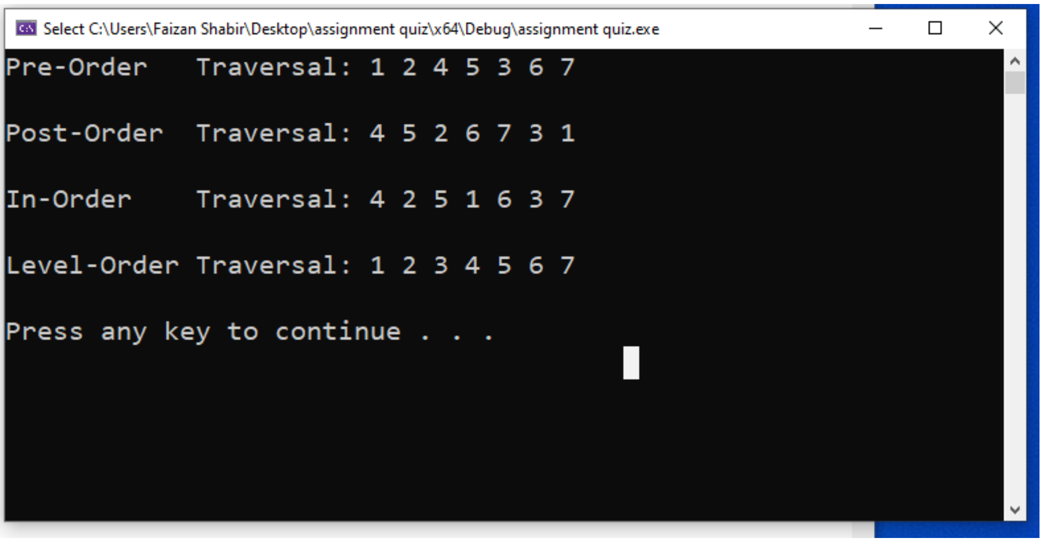
```cpp
// Level-Order Iterator
class TreeLevelOrderIterator : public Iterator
{
    queue<Node*> Queue;
public:
    TreeLevelOrderIterator(Node * root)
    {
        if(root!=nullptr)
        {
            Queue.push(root);
        }
    }

    bool hasNext()
    {
        return (!Queue.empty());
    }

    int Next()
    {
        Node* current = Queue.front(); // get current node from queue
        Queue.pop();

        if(current->left) // push left child in queue if any
        {
            Queue.push(current->left);
        }

        if(current->right) // push right child in queue if any
        {
            Queue.push(current->right);
        }

        return current->value; // return current node's value
    }
};
///////////////////////////////////////////////////////////////
```

```cpp
int main()
{
      // create a small binary tree manually
      Node* root = new Node(1);         ///         1
      root->left = new Node(2);         ///        / \
      root->right = new Node(3);        ///       2   3
      root->left->left = new Node(4);   ////     / \ / \
      root->left->right = new Node(5);  ///      4 5 6 7
      root->right->left = new Node(6);  ///
      root->right->right = new Node(7); ///
      ///////////////////////////////////////////////////////////
      // create and test Pre-Order Iterator
      Iterator * iterator = new TreePreOrderIterator(root);
      cout << "Pre-Order   Traversal: ";
      while(iterator->hasNext())
      {
            cout << iterator->Next() << " ";
      }
      cout << endl << endl;
      delete iterator;
      ///////////////////////////////////////////////////////////
      // create and test Post-Order Iterator
      iterator = new TreePostOrderIterator(root);
      cout << "Post-Order  Traversal: ";
      while (iterator->hasNext())
      {
            cout << iterator->Next() << " ";
      }
      cout << endl << endl;
      delete iterator;
      ///////////////////////////////////////////////////////////
      // create and test In-Order Iterator
      iterator = new TreeInOrderIterator(root);
      cout << "In-Order    Traversal: ";
      while (iterator->hasNext())
      {
            cout << iterator->Next() << " ";
      }
      cout << endl << endl;
      delete iterator;
      ///////////////////////////////////////////////////////////
      // create and test Level-Order Iterator
      iterator = new TreeLevelOrderIterator(root);
      cout << "Level-Order Traversal: ";
      while (iterator->hasNext())
      {
            cout << iterator->Next() << " ";
      }
      cout << endl << endl;
      delete iterator; // delete iterator
      ///////////////////////////////////////////////////////////
      // delete the manually create binary tree
      delete root->right->right;
      delete root->right->left;
      delete root->left->right;
      delete root->left->left;
      delete root->right;
      delete root->left;
      delete root;
      system("pause");
}
```

Code Output Screenshot



Binary Tree Data Structure


Select C:\Users\Faizan Shabir\Desktop\assignment quiz\x64\Debug\assignment quiz.exe

```
Pre-Order   Traversal: 1 2 4 5 3 6 7

Post-Order  Traversal: 4 5 2 6 7 3 1

In-Order    Traversal: 4 2 5 1 6 3 7

Level-Order Traversal: 1 2 3 4 5 6 7

Press any key to continue . . .
```