TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP. HỒ CHÍ MINH

KHOA ĐÀO TẠO CHẤT LƯỢNG CAO

BÁO CÁO ĐỒ ÁN 1

TÌM HIỂU THUẬT TOÁN BST VÀ AVL TREE
SO SÁNH SỰ GIỐNG VÀ KHÁC NHAU, CÀI ĐẶT MINH HỌA BẰNG GIAO DIỆN, VÀ CHẠY CHO DỮ LIỆU THỰC CHO CẢ 2 THUẬT TOÁN TRÊN

GVHD : **TS. Lê Văn Vinh**

SVTH :

Đoàn Quốc Việt **19110314**

Lớp 19110CL5

Nguyễn Lê Bảo Thanh **19110019**

Lớp 19110CL1

Tp. Hồ Chí Minh, tháng 12 năm 2021

LỜI CẢM ƠN

Để hoàn thành tốt đề tài và bài báo cáo này, chúng em xin gửi lời cảm ơn chân thành đến giảng viên, tiến sĩ Lê Văn Vinh, người đã trực tiếp hỗ trợ chúng em trong suốt quá trình làm đề tài. Chúng em cảm ơn thầy đã đưa ra những lời khuyên từ kinh nghiệm thực tiễn của mình để định hướng cho chúng em đi đúng với yêu cầu của đề tài đã chọn, luôn giải đáp thắc mắc và đưa ra những góp ý, chỉnh sửa kịp thời giúp chúng em khắc phục nhược điểm và hoàn thành tốt cũng như đúng thời hạn đã đề ra.

Chúng em cũng xin gửi lời cảm ơn chân thành các quý thầy cô trong khoa Đào tạo Chất Lượng Cao nói chung và ngành Công Nghệ Thông Tin nói riêng đã tận tình truyền đạt những kiến thức cần thiết giúp chúng em có nền tảng để làm nên đề tài này, đã tạo điều kiện để chúng em có thể tìm hiểu và thực hiện tốt đề tài. Cùng với đó, chúng em xin được gửi cảm ơn đến các bạn cùng khóa đã cung cấp nhiều thông tin và kiến thức hữu ích giúp chúng em có thể hoàn thiện hơn đề tài của mình.

Đề tài và bài báo cáo được chúng em thực hiện trong khoảng thời gian ngắn, với những kiến thức còn hạn chế cùng nhiều hạn chế khác về mặt kĩ thuật và kinh nghiệm trong việc thực hiện một dự án phần mềm. Do đó, trong quá trình làm nên đề tài có những thiếu sót là điều không thể tránh khỏi nên chúng em rất mong nhận được những ý kiến đóng góp quý báu của các quý thầy cô để kiến thức của chúng em được hoàn thiện hơn và chúng em có thể làm tốt hơn nữa trong những lần sau.

Chúng em xin chân thành cảm ơn. Cuối lời, chúng em kính chúc quý thầy, quý cô luôn dồi dào sức khỏe và thành công hơn nữa trong sự nghiệp trồng người. Một lần nữa chúng em xin chân thành cảm ơn.

Tp. Hồ Chí Minh, ngày tháng 12 năm 2021

Nhóm sinh viên thực hiện

MỤC LỤC

DANH MỤC HÌNH ẢNH

DANH MỤC BẢNG

Bảng 1 – Lên kế hoạch

Tên sinh viên	Thời gian thực hiện	Tiến độ hoàn thành	Nhiệm vụ
Nguyễn Lê Bảo Thanh Đoàn Quốc Việt	1/9/2021 đến 8/9/2021	100%	Tìm hiểu lý thuyết về 2 thuật toán BST và AVL Tree như: - Định nghĩa - Cách thức hoạt động - Ứng dụng - Phân tích, so sánh ưu và nhược điểm của 2 thuật toán
Đoàn Quốc Việt	9/9/2021 đến 21/10/2021	100%	Viết code và làm giao diện cho thuật toán BST
Nguyễn Lê Bảo Thanh	9/9/2021 đến 24/10/2021	100%	Viết code và làm giao diện cho thuật toán AVL Tree
Nguyễn Lê Bảo Thanh Đoàn Quốc Việt	25/10/2021 đến 28/11/2021	100%	Chỉnh sửa lỗi cho đồ án
Nguyễn Lê Bảo Thanh	29/11/2021 đến 1/12/2021	100%	Viết báo cáo cho các phần lý thuyết của 2 thuật toán
Đoàn Quốc Việt	29/11/2021 đến 1/12/2021	100%	Viết báo cáo phần phân tích đồ án

DANH MỤC CÁC TỪ VIẾT TẮT

- BST: Binary Search Tree.
- AVL: viết tắt của tên các nhà phát minh Adelson, Velski và Landis.
- CTDL: Cấu trúc dữ liệu.
- UML: Unified Modeling Language

CHƯƠNG 1: KHÁI NIỆM CHUNG VỀ CTDL CÂY

1.1. Cây là gì?

Trong khoa học máy tính, cây là một cấu trúc dữ liệu được sử dụng rộng rãi gồm một tập hợp các node (tiếng Anh: node) được liên kết với nhau theo quan hệ cha-con. Cây trong cấu trúc dữ liệu đầu tiên là mô phỏng (hay nói cách khác là sự sao chép) của cây (có gốc) trong lý thuyết đồ thị. Hầu như mọi khái niệm trong cây của lý thuyết đồ thị đều được thể hiện trong cấu trúc dữ liệu. Tuy nhiên cây trong cấu trúc dữ liệu đã tìm được ứng dụng phong phú và hiệu quả trong nhiều giải thuật. Khi phân tích các giải thuật trên cấu trúc dữ liệu cây, người ta vẫn thường vẽ ra các cây tương ứng trong lý thuyết đồ thị.

1.2. Các loại CTDL cây

1.2.1. Cây tìm kiếm (Search Tree)

Được sử dụng để định vị các khóa cụ thể từ trong một tập hợp. Để một cây hoạt động như một cây tìm kiếm, khóa cho mỗi node phải lớn hơn bất kỳ khóa nào trong cây con ở bên trái và nhỏ hơn bất kỳ khóa nào trong cây con ở bên phải.

- 2–3
- 2–3–4
- AA
- (a,b)
- **AVL**
- B
- B+
- B*
- B^x
- **(Optimal) Binary search**
- Dancing
- HTree
- Interval
- Order statistic
- (Left-leaning) Red–black
- Scapegoat
- Splay
- T
- Treap
- UB
- Weight-balanced

1.2.2. Đống (Heap)

Là một cấu trúc dữ liệu dựa trên cây thỏa mãn tính chất đống: nếu B là node con của A thì khóa(A)≥khóa(B). Một hệ quả của tính chất này là khóa lớn nhất luôn nằm ở node gốc. Do đó một đống như vậy thường được gọi là max-heap. Nếu mọi phép so sánh bị đảo ngược khiến cho khóa nhỏ nhất luôn nằm ở node gốc thì đống đó gọi là min-heap. Không có hạn chế nào về số lượng node con của mỗi node trong đống nhưng thông thường mỗi node có không quá hai node con. Đống là một cách thực hiện kiểu dữ liệu trừu tượng mang tên hàng đợi ưu tiên. Đống có vai trò quan trọng trong nhiều thuật toán cho đồ thị chẳng hạn như thuật toán Dijkstra, hay thuật toán sắp xếp heapsort.

- Binary
- Binomial
- Brodal
- Fibonacci
- Leftist
- Pairing
- Skew
- van Emde Boas
- Weak

1.2.3. Trie (cây tiền tố)

Là một cấu trúc dữ liệu sử dụng cây có thứ tự, dùng để lưu trữ một mảng liên kết của các xâu ký tự. Không như cây nhị phân tìm kiếm, mỗi node trong cây không liên kết với một khóa trong mảng. Thay vào đó, mỗi node liên kết với một xâu ký tự sao cho các xâu ký tự của tất cả các node con của một node đều có chung một tiền tố, chính là xâu ký tự của node đó. Node gốc tương ứng với xâu ký tự rỗng.

- Ctrie
- C-trie (compressed ADT)
- Hash
- Radix
- Suffix
- Ternary search
- X-fast
- Y-fast

1.2.4. Spatial data partitioning trees

- Ball
- BK
- BSP
- Cartesian
- Hilbert R
- k-d (implicit k-d)
- M
- Metric
- MVP
- Octree
- Priority R
- Quad
- R
- R+
- R*
- Segment
- VP
- X

1.2.5. Khác

- Cover
- Exponential
- Fenwick
- Finger
- Fractal tree index
- Fusion
- Hash calendar
- iDistance
- K-ary
- Left-child right-sibling
- Link/cut
- Log-structured merge
- Merkle
- PQ
- Range
- SPQR
- Top

1.3. Thuật toán

1.3.1. Tạo cây

- Thuật toán BST
- Thuật toán AVLT

1.3.2. Các phương pháp duyệt cây

- Duyệt tiền thứ tự
- Duyệt trung thứ tự
- Duyệt hậu thứ tự

1.3.3. Các giải thuật chung

- Tìm kiếm một mục trên cây

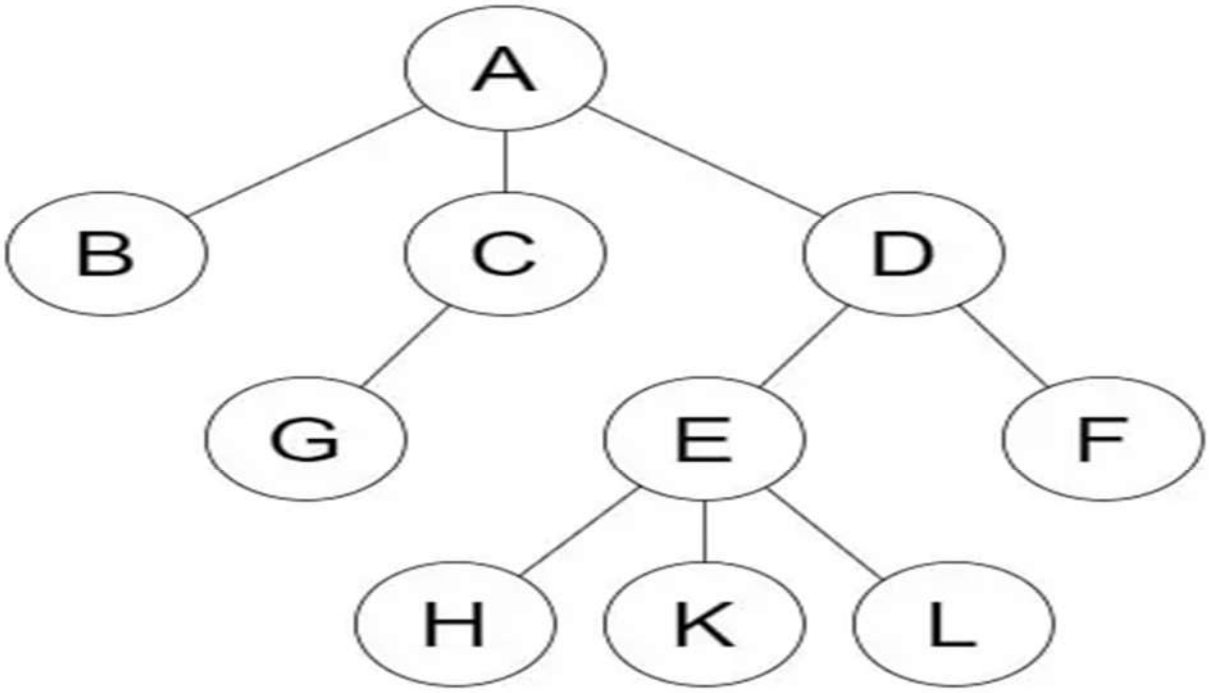
- Bổ sung một mục mới
- Xóa một mục

CHƯƠNG 2: TÌM HIỂU BINARY SEARCH TREE (CÂY TÌM KIẾM NHỊ PHÂN)

2.1. Cây tìm kiếm nhị phân là gì?

Cây tìm kiếm nhị phân(**B**inary **S**earch **T**ree – viết tắt: **BST**) – là một cây nhị phân và có thêm các ràng buộc sau đây:

1. Giá trị của tất cả các Node ở cây con bên trái phải <= giá trị của Node gốc.
2. Giá trị của tất cả các Node ở cây con bên phải phải > giá trị của Node gốc.
3. Tất cả các cây con(bao gồm bên trái và phải) cũng đều phải đảm bảo 2 tính chất trên.



Hình 1 - Hình dạng của cây nhị phân

2.2. Các thuật ngữ

- Bậc của node: là số node con của node đó. Ví dụ bậc của node A là 3, bậc của node C là 1, bậc của node G là 0…

- Bậc của cây: là bậc lớn nhất của node trong cây đó, cây bậc n sẽ được gọi là cây n – phân. Ví dụ cây trong hình trên có bậc 3, gọi là cây tam phân, cây có bậc 2 gọi là cây nhị phân…
- Node lá: node lá là node có bậc bằng 0. Ví dụ các node lá: B, G, H, K, L, F
- Node nhánh: là node có bậc khác 0 mà không phải node gốc (hay còn gọi là node trung gian). Ví dụ các node C, D, E
- Mức của node: là số nguyên đếm từ 0, các node ngang hàng nhau thì có cùng mức. Node gốc A có mức là 0, mức 1 gồm các node B, C, D, node 3 gồm H, K, L. Chiều cao (chiều sâu): là mức lớn nhất của các node lá. Ví dụ cây trên có node lá bậc lớn nhất là H, K, L mức 3, vậy chiều cao của cây là 3.
- Độ dài đường đi đến node x: là số nhánh (cạnh nối hai node) cần đi qua tính từ node gốc đến node x. Hay độ dài đường đi đến node mức i chính là i. Ví dụ node E có độ dài đường đi là 2.

2.3. Thuật toán (pseudocode – mã giả)

2.3.1. Tạo cây

```
struct Node
{
        int data;
        Node* right;
        Node* left;
};
typedef struct Node* tree;
```

2.3.2. Thuật toán search

*Ý tưởng:

1. Nếu cây rỗng (null): không tìm thấy
2. Nếu key == node.key: tìm thấy

3. Nếu key < node.key: gọi hàm search cây con trái của node
4. Nếu key > node.key: gọi hàm search cây con phải của node

```
if node is Null then
       return None;  /* key not found */

      if key < node.key:
          return search binary_tree(node.left, key);
      else
          if key > node.key
              return search_binary_tree(node.right, key)
          else  /* key is equal to node key */
              return node.value;  # found key
```

2.3.3. Thuật toán insert

*Ý tưởng:

1. Nếu cây rỗng (null): tạo cây (insert key vào root)
2. Nếu key không rỗng:
   2.1. Nếu key < node.key: gọi hàm insert cây con trái của node
   2.2. Nếu key > node.key: gọi hàm insert cây con phải của node

```
void InsertNode(struct node *&treeNode, struct node *newNode)
{ //Inserts node pointered by "newNode" to the subtree started
by "treeNode"
    if (treeNode == NULL)
        treeNode = newNode; //Only changes "node" when it is
NULL
    else if (newNode->value < treeNode->value)
```

```
        InsertNode(treeNode->left, newNode);
    else
        InsertNode(treeNode->right, newNode);
}
```

2.3.4. Thuật toán delete

*Ý tưởng:

1. Nếu xóa node lá: giải phóng nó ra khỏi cây
2. Nếu không phải là node lá:

   - Xóa node có một con: Xóa và thay thế nó bằng con duy nhất của nó.

   - Xóa một node có hai con: Xóa node đó và thay thế nó bằng node có khóa lớn nhất trong các khóa nhỏ hơn khóa của nó (được gọi là "node tiền nhiệm" -node cực phải của cây con trái) hoặc node có khóa nhỏ nhất trong các khóa lớn hơn nó (được gọi là "node kế vị" - node cực trái của cây con phải)

```
void DeleteNode(struct node*& node) {
    if (node->left == NULL) {

        struct node* temp = node;
        node = node->right;
        delete temp;
    } else if (node->right == NULL) {

        struct node* temp = node;
```

```
        node = node->left;
        delete temp;

    } else {
        // In-Order predecessor(right most child of left subtree)
        // Node has two children - get max of left subtree

        struct node** temp = &(node->left); // get left node of the original node

        // find the right most child of the subtree of the left node
        while ((*temp)->right != NULL) {
            temp = &((*temp)->right);
        }

        // copy the value from the right most child of left subtree to the original node
        node->value = (*temp)->value;

        // then delete the right most child of left subtree since it's value is
        // now in the original node
```

```
            DeleteNode(*temp);
        }
    }
```

2.3.5. Các phương pháp duyệt cây (Code C++ Visual Studio 2019)

1. Left-Node-Right (LNR, duyệt trung(giữa))

```
void LNR(tree& t)
{
    if (t)
    {
        LNR(t->left);
        cout << t->data << " ";
        LNR(t->right);
    }
}
```

Hình 2 - Duyệt LNR

2. Right-Node-Left (RNL)

```
void RNL(tree& t)
{
    if (t)
    {
        RNL(t->right);
        cout << t->data << " ";
        RNL(t->left);
    }
}
```

Hình 3 - Duyệt RNL

3. Left-Right-Node (LRN, duyệt sau)

```
void LRN(tree& t)
{
    if (t)
    {
        LRN(t->left);
        LRN(t->right);
        cout << t->data << " ";
    }
}
```

Hình 4– Duyệt LRN

4. Right-Left-Node (RLN)

```
void RLN(tree& t)
{
    if (t)
    {
        RLN(t->right);
        RLN(t->left);
        cout << t->data << " ";
    }
}
```

Hình 5– Duyệt RLN

5. Node-Right-Left (NRL)

```
void NRL(tree& t)
{
    if (t)
    {
        cout << t->data << " ";
        NRL(t->right);
        NRL(t->left);
    }
}
```

Hình 6 – Duyệt NRL

6. Node-Left-Right (NLR, duyệt trước)

```
void NLR(tree& t)
{
    if (t)
    {
        cout << t->data << " ";
        NLR(t->left);
        NLR(t->right);
    }
}
```

Hình 7– Duyệt NLR

2.3. Ứng dụng

- Các BST được sử dụng để lập chỉ mục và lập chỉ mục nhiều cấp.
- Chúng cũng hữu ích để triển khai các thuật toán tìm kiếm khác nhau.
- Nó rất hữu ích trong việc duy trì luồng dữ liệu được sắp xếp.

2.4. Ưu nhược điểm

2.4.1. Ưu điểm

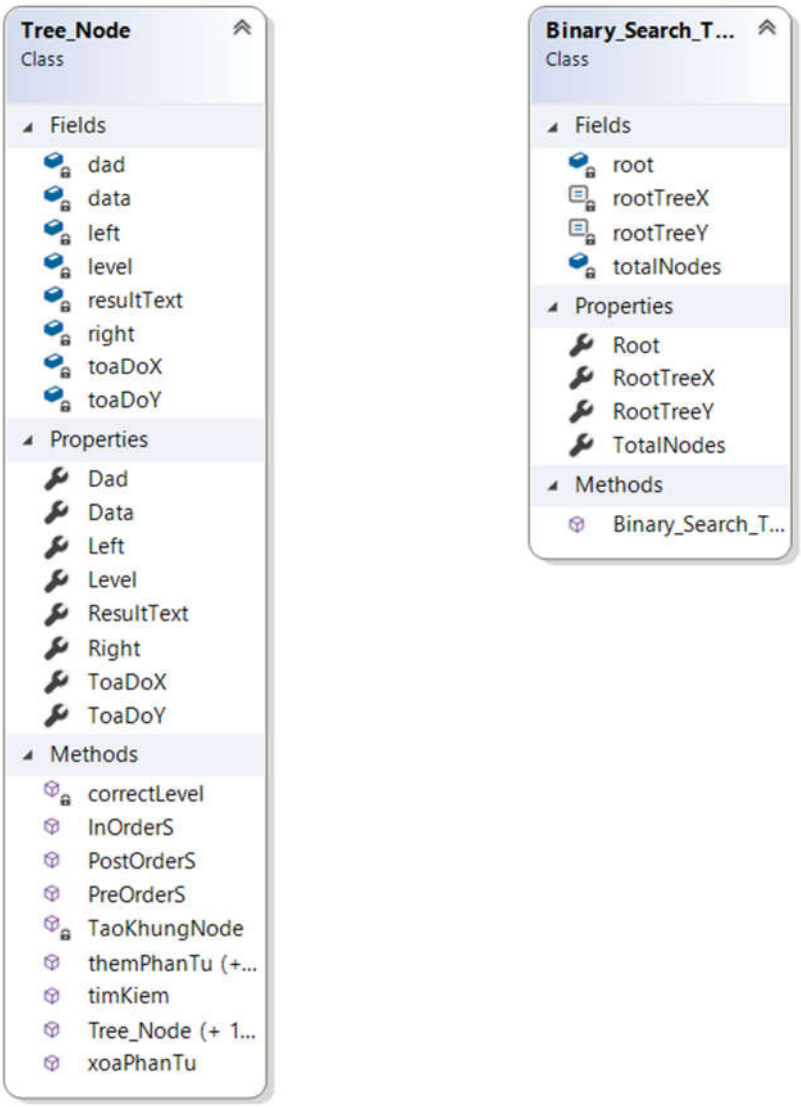
- Chúng ta luôn có thể giữ chi phí của insert (), delete (), lookup () thành O (logN) trong đó N là số nút trong cây - vì vậy lợi ích thực sự là việc tra cứu có thể được thực hiện theo thời gian logarit, điều này quan trọng khá nhiều khi N lớn.
- Chúng ta có một thứ tự các chìa khóa được lưu trữ trên cây. Bất kỳ lúc nào chúng ta cần duyệt theo thứ tự tăng (hoặc giảm) của các phím, chúng ta chỉ cần thực hiện việc di chuyển theo thứ tự (và đảo ngược thứ tự) trên cây.
- Chúng ta có thể thực hiện thống kê thứ tự với cây tìm kiếm nhị phân - phần tử nhỏ nhất thứ N, phần tử lớn thứ n. Điều này là do có thể xem cấu trúc dữ liệu như một mảng được sắp xếp.
- Chúng ta cũng có thể thực hiện các truy vấn phạm vi - tìm các khóa giữa N và M (N <= M).
- BST cũng có thể được sử dụng trong việc thiết kế trình cấp phát bộ nhớ để tăng tốc độ tìm kiếm các khối trống (khối bộ nhớ) và triển khai các thuật toán phù hợp nhất mà chúng tôi quan tâm đến việc tìm kiếm khối trống nhỏ nhất có kích thước lớn hơn hoặc bằng kích thước được chỉ định trong yêu cầu phân bổ.

2.4.2. Nhược điểm

- Cây sẽ mất cân bằng (các node lệch hẳn về một bên) khi dữ liệu vào là tăng/giảm dần.
- Có thể biến thành danh sách liên kết.

CHƯƠNG 3: MINH HỌA THUẬT TOÁN BINARY SEARCH TREE (CÂY TÌM KIẾM NHỊ PHÂN)

3.1. Thiết kế



Hình 8– Mô hình UML của ứng dụng BST

- Class **Tree_Node:**

  Fields:

  dad: node cha của node hiện tại.

  data: giá trị của node hiện tại.

  left, right: node con trái và phải của node hiện tại.

level: bậc của node hiện tại.

toaDoX, toaDoY: vị trí của node hiện tại để biểu diễn.

Method:

correctLevel: chỉnh lại bậc của từng node cho hợp lệ.

InOrderS: duyệt giữa.

PosOrderS: duyệt sau.

PreOrderS: duyệt trước.

TaoKhungNode: dùng để vẽ khung khi tìm hoặc duyệt node.

ThemPhanTu: thêm phần tử do người dùng nhập (có overloading).

timKiem: tìm phần tử do người dùng nhập.

Tree_Node: constructor của class (có overloading)

xoaPhanTu: xóa phần tử do người dùng nhập.

- Class **Binary_Search_Tree:**

Fields:

root: node gốc của cây.

rootTreeX, rootTreeY: tọa độ của node gốc của cây.

totalNodes: tổng số node trong cây.

Method:

Binary_Tree: constructor của class.

3.2. Code

3.2.1. Tree_Node.cs

```
class Tree_Node
{

    int data;  // giá trị của node hiện tại
    int level; // bậc của node hiện tại
    Tree_Node left; // node con trái của node hiện tại
    Tree_Node right;// node con phải của node hiện tại
    Tree_Node dad; // node cha của node hiện tại

    // vị trí của node hiện tại để biểu diễn
    int toaDoX;
    int toaDoY;
    string resultText = "";
```

Hình 9– Khai báo các biến cần thiết

```
public Tree_Node(Tree_Node tree)
{
    data = 0; // giá trị của node hiện tại là 0
    dad = tree; // gán node cha của node đó = tree
    level = tree.level + 1; // tăng bậc của node hiện tại lên 1
}
```

Hình 10– Hàm tạo Tree_Node

```
public void themPhanTu(ref Tree_Node tree, int data, ref int canAdd)
{
    if (tree.data == 0)
    {
        tree.data = data;
        canAdd = 1;
    }
    else  //data của tree đã có giá trị nên duyệt xuống các cây con
    {
        if (data < tree.data)  //đi qua trái
        {
            if (tree.left == null) //khởi tạo node con trái
            {
                Tree_Node tmp = new Tree_Node(tree);
                tree.left = tmp;
            }
            themPhanTu(ref tree.left, data, ref canAdd);
        }
        else if (data > tree.data)  //đi qua phải
        {
            if (tree.right == null) //khởi tạo node con phải
            {
                Tree_Node tmp = new Tree_Node(tree);
                tree.right = tmp;
            }
            themPhanTu(ref tree.right, data, ref canAdd);
        }
        else  //data == tree.data
        {
            canAdd = 0;
            MessageBox.Show($"Node { data } already in tree!", "Warning!", MessageBoxButtons.OK, MessageBoxIcon.Warning);
        }
    }
} //của add button
```

Hình 11– Hàm themPhanTu (Chức năng Add)

```
public void themPhanTu(ref Tree_Node tree, int data)
{
    if (tree.data == 0)
    {
        tree.data = data;
    }
    else  //data của tree đã có giá trị nên duyệt xuống các cây con
    {
        if (data < tree.data)  //đi qua trái
        {
            if (tree.left == null) //khởi tạo node con trái
            {
                Tree_Node tmp = new Tree_Node(tree);
                tree.left = tmp;
            }
            themPhanTu(ref tree.left, data);
        }
        else if (data > tree.data)  //đi qua phải
        {
            if (tree.right == null) //khởi tạo node con phải
            {
                Tree_Node tmp = new Tree_Node(tree);
                tree.right = tmp;
            }
            themPhanTu(ref tree.right, data);
        }
    }
} //của random new tree button
```

Hình 12– Hàm themPhanTu (Chức năng Random)

```
public async void timKiem(Tree_Node tree, int data, Form2 form1)
{
    if (tree == null) // không tìm thấy
    {
        MessageBox.Show($"Tree don't have node { data }", "Result", MessageBoxButtons.OK, MessageBoxIcon.Warning);
    }
    else
    {
        if (tree.data != data) // chưa tìm thấy
        {
            TaoKhungNode(tree, Color.Red, form1);
            //form1.treeArea.Controls.Add(lbl);
            await Task.Delay(500);
        }
        else // tìm thấy
        {
            TaoKhungNode(tree, Color.Green, form1);
            //form1.treeArea.Controls.Add(lbl);
            var result = MessageBox.Show($"Found node {data}", "Result", MessageBoxButtons.OK, MessageBoxIcon.Information);
            if(result == DialogResult.OK)
            {
                form1.treeArea.Invalidate();
            }
        }
        if (data < tree.data) // nếu dữ liệu cần tìm nhỏ hơn dữ liệu nút đang xét
        {
            timKiem(tree.left, data, form1);
        }

        else if (data > tree.data) // nếu dữ liệu cần tìm lớn hơn dữ liệu nút đang xét
        {
            timKiem(tree.right, data, form1);
        }
    }
}
```

Hình 13– Hàm timKiem (Chức năng Find)

```
public void xoaPhanTu(ref Tree_Node tree, ref int canDel, int data)
{
    if (tree == null)
    {
        MessageBox.Show($"Tree don't have node { data } to delete!", "Error!", MessageBoxButtons.OK, MessageBoxIcon.Error);
    }

    else
    {
        if (data < tree.data) // đi tìm nút cần xóa
        {
            xoaPhanTu(ref tree.left, ref canDel, data);
        }

        else if (data > tree.data) // đi tìm nút cần xóa
        {
            xoaPhanTu(ref tree.right, ref canDel, data);
        }

        else  // tìm thấy nút cần xóa
        {
            if (tree.left == null && tree.right == null) //nút cần xóa là lá
            {
                if (tree.level != 0) //không phải root
                {
                    if (tree.dad.left != null && tree.dad.left.data == tree.data) //kiểm tra tree là con phải hay con trái của tree.dad
                        tree.dad.left = null;
                    else
                        tree.dad.right = null;
                }
                else //là root
                {
                    canDel = 2;
                    return;
                }
                canDel = 1;
            }
            else if (tree.left == null)  // nút cần xóa chỉ có nút phải
            {
                if (tree.level != 0) //không phải root
                {
                    tree.right.dad = tree.dad; //chuyển cha của nút phải thành cha của nút hiện tại
```

Hình 14– Hàm xoaPhanTu (Chức năng Delete)

```
public void PreOrderS(Tree_Node tree, Form2 form1, int delay) // duyệt trước
{
    if (tree != null)
    {
        TaoKhungNode(tree, Color.Green, form1);
        resultText += resultText.Equals("") ? tree.data.ToString() : $", {tree.data}";
        //form1.Resulttxt.Text = resultText;
        Thread.Sleep(delay);
        PreOrderS(tree.left, form1, delay);
        PreOrderS(tree.right, form1, delay);

    }
}

public void PostOrderS(Tree_Node tree, Form2 form1, int delay) // duyệt giữa
{
    if (tree != null)
    {
        PostOrderS(tree.left, form1, delay);
        PostOrderS(tree.right, form1, delay);
        TaoKhungNode(tree, Color.Green, form1);
        resultText += resultText.Equals("") ? tree.data.ToString() : $", {tree.data}";
        //form1.Resulttxt.Text = resultText;
        Thread.Sleep(delay);
    }
}

public void InOrderS(Tree_Node tree, Form2 form1, int delay) // duyệt sau
{
    if (tree != null)
    {
        InOrderS(tree.left, form1, delay);
        TaoKhungNode(tree, Color.Green, form1);
        resultText += resultText.Equals("") ? tree.data.ToString() : $", {tree.data}";
        //form1.Resulttxt.Text = resultText;
        Thread.Sleep(delay);
        InOrderS(tree.right, form1, delay);
    }
}
```

Hình 15 – Các hàm PreOrderS (duyệt trước), PostOrderS (duyệt giữa), InOrderS (duyệt sau)

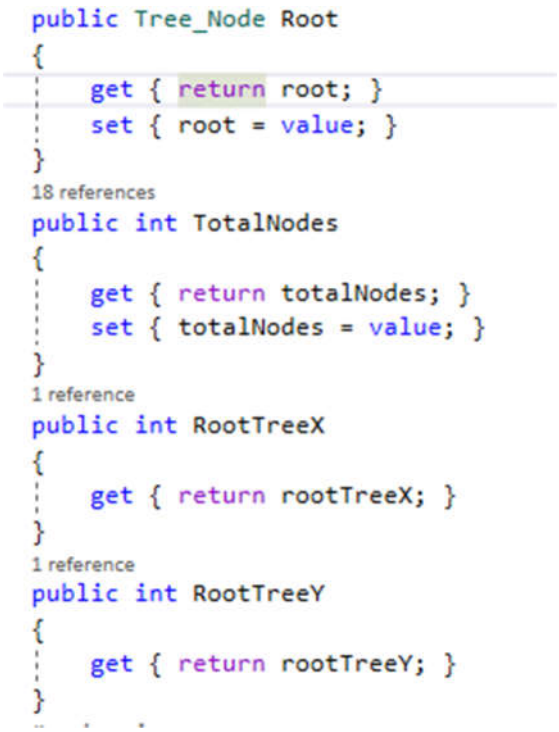
```
private void correctLevel(ref Tree_Node tree) // so sánh và xác định node thêm vào sẽ sang trái hay sang phải của node đang xét
{
    if (tree.level != 0)
        tree.level = tree.dad.level + 1;

    if (tree.left != null)
        correctLevel(ref tree.left);
    if (tree.right != null)
        correctLevel(ref tree.right);
}

private void TaoKhungNode(Tree_Node tmp, Color color, Form2 form1) // tạo khung cho node
{
    SolidBrush mybrush = new SolidBrush(color);
    Pen mypen = new Pen(mybrush, 7);
    Graphics g = form1.treeArea.CreateGraphics();
    g.DrawRectangle(mypen, tmp.toaDoX - 1, tmp.toaDoY - 1, form1.NodeSize + 2, form1.NodeSize + 2);
}
```

Hình 16– Hàm correctLevel và hàm TaoKhungNode

3.2.2. Binary_Search_Tree.cs

```
class Binary_Search_Tree
{
    Tree_Node root;
    int totalNodes = 0;
    const int rootTreeX = 50; //tọa độ node đầu tiên
    const int rootTreeY = 15;

    4 references
    public Binary_Search_Tree()
    {
        Tree_Node tmp = new Tree_Node();
        tmp.Data = 0;
        tmp.Level = 0;
        tmp.ToaDoX = rootTreeX;
        tmp.ToaDoY = rootTreeY;
        root = tmp;
    }
```

Hình 17– Khai báo các biến cần thiết, hàm tạo Binary_Search_Tree

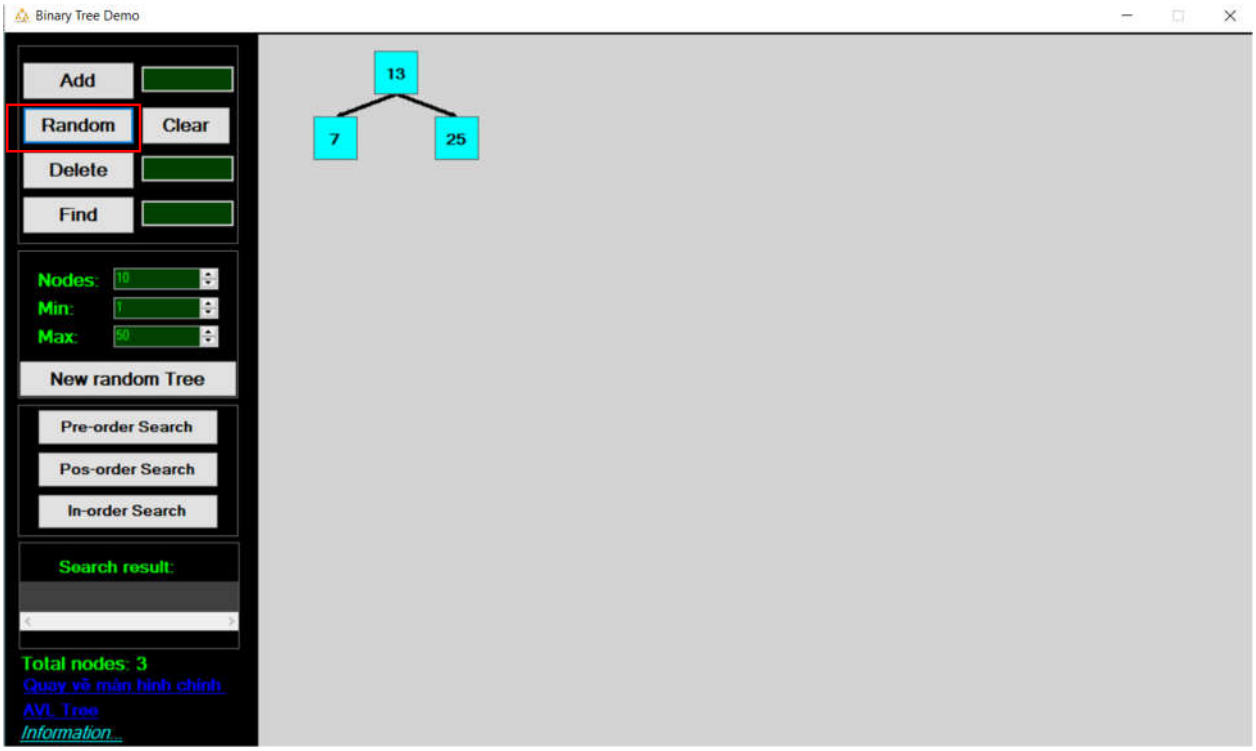


Hình 18– Các phương thức

3.3. Giao diện

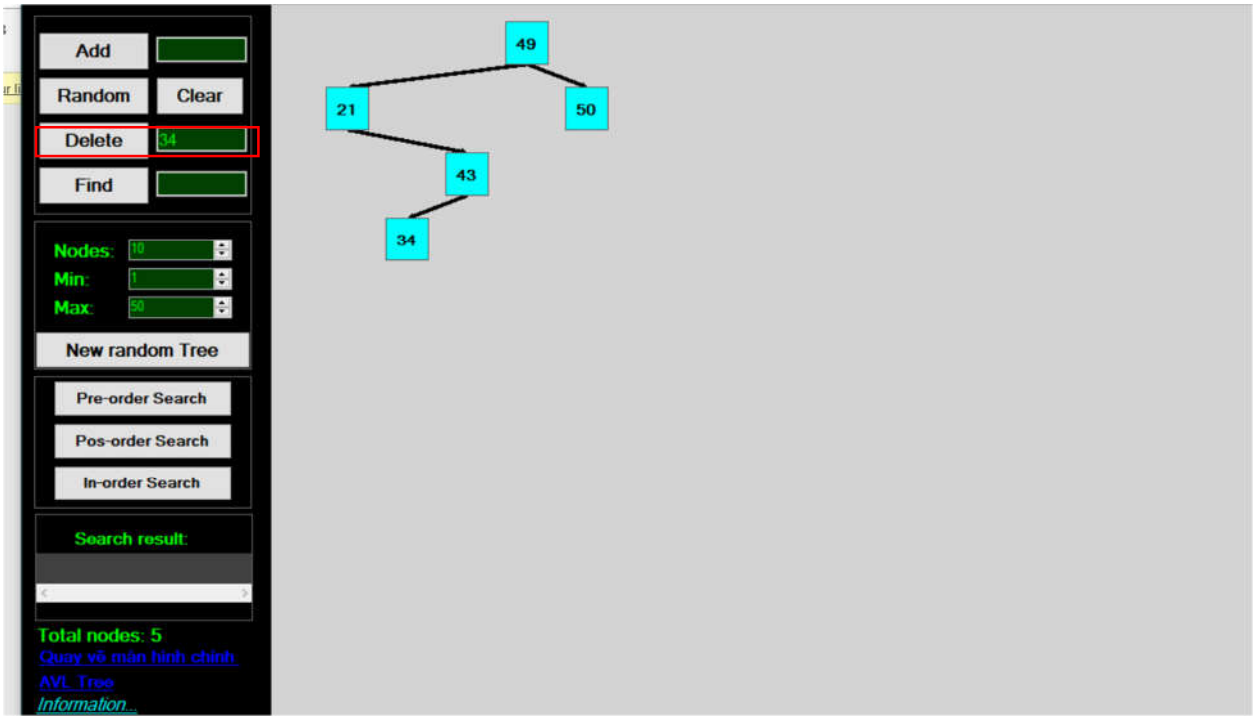
Hình 19– Giao diện chính của ứng dụng demo BST

Hình 20– Thêm node giá trị 13 bằng nút Add

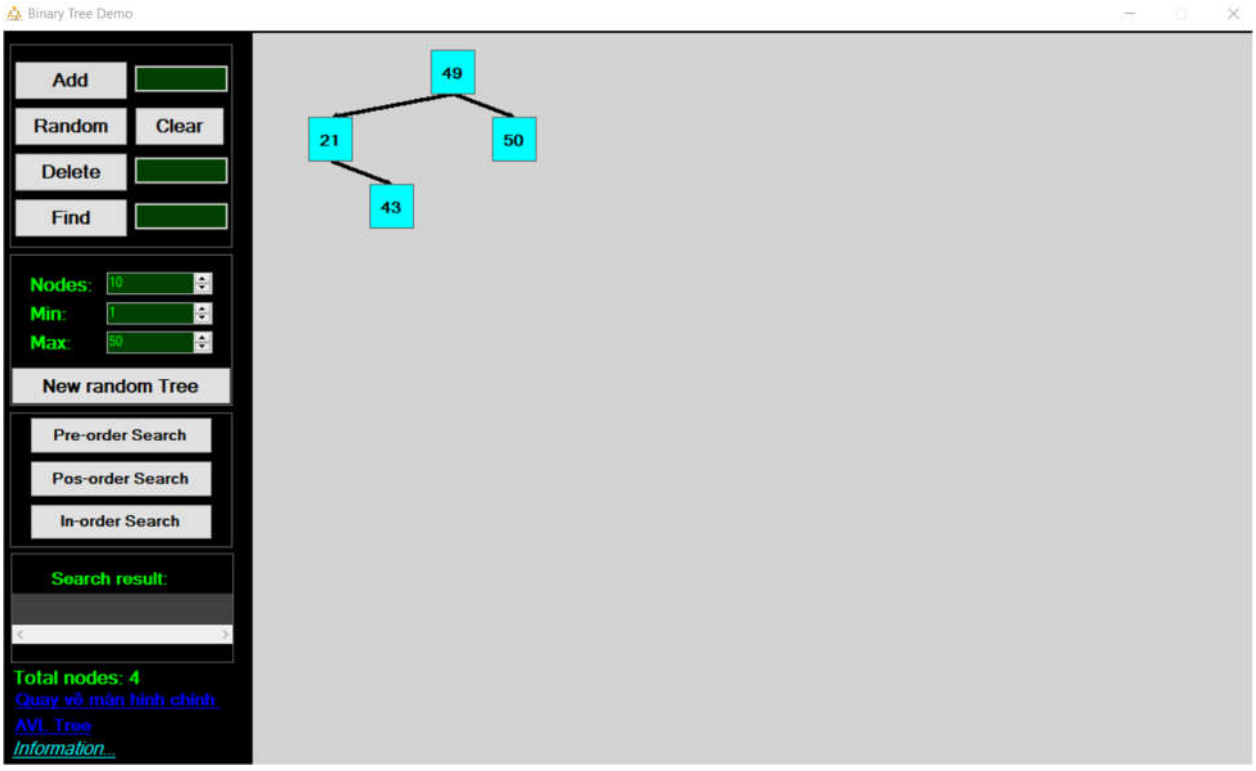


Hình 21 – Nhấn nút Random 2 lần thì hai node 7 và 25 được thêm vào

Hình 22– Nhấn nút Clear để xóa toàn bộ các node

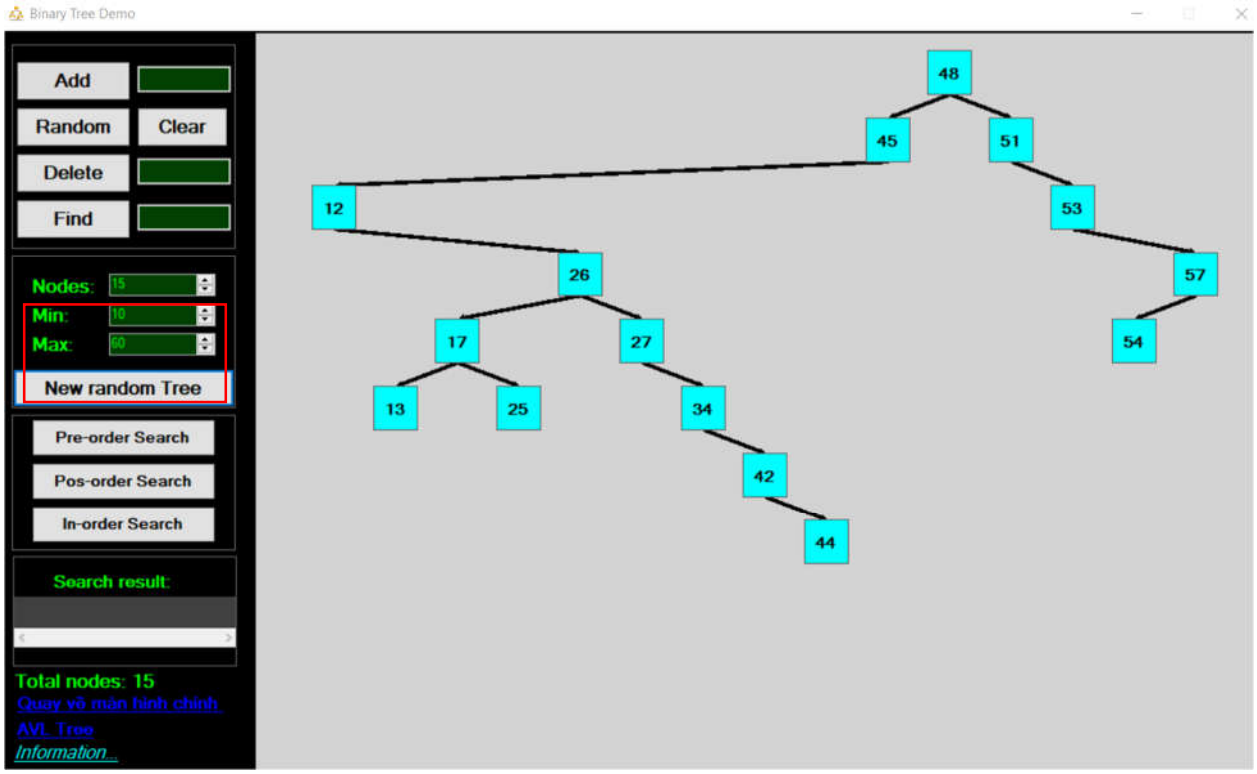


Hình 23– Nhập giá trị cần xóa và nhấn Delete để xóa Node đó

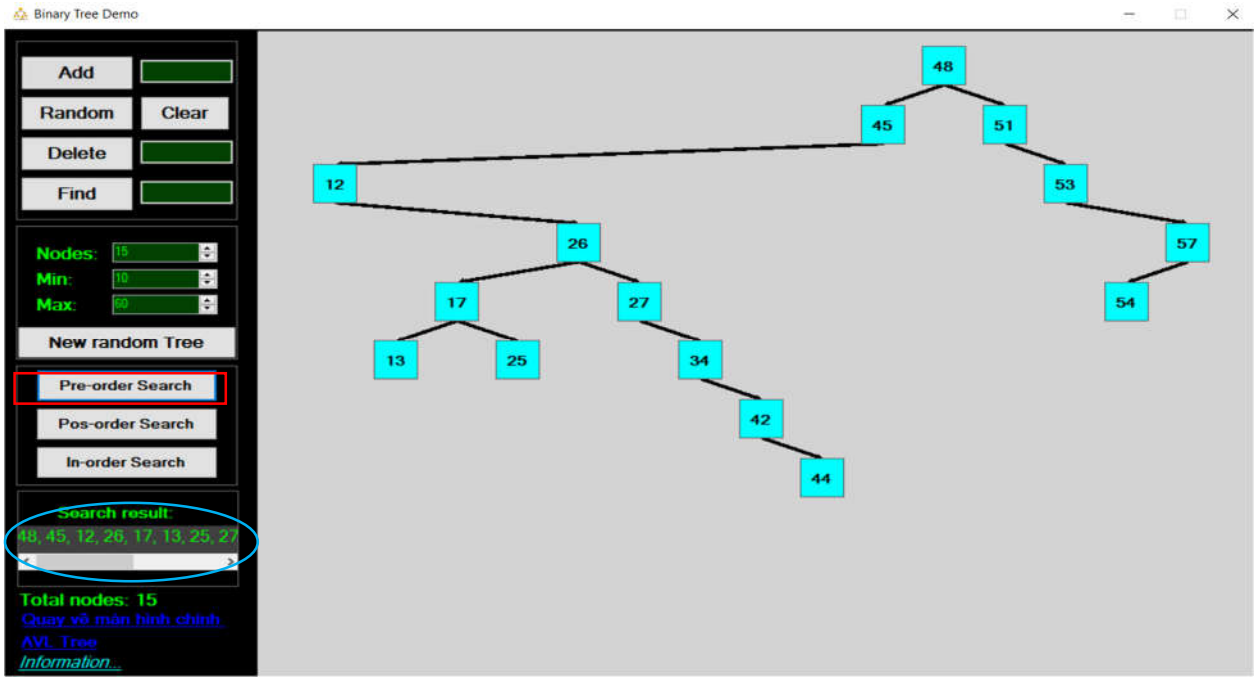


Hình 24– Kết quả Delete

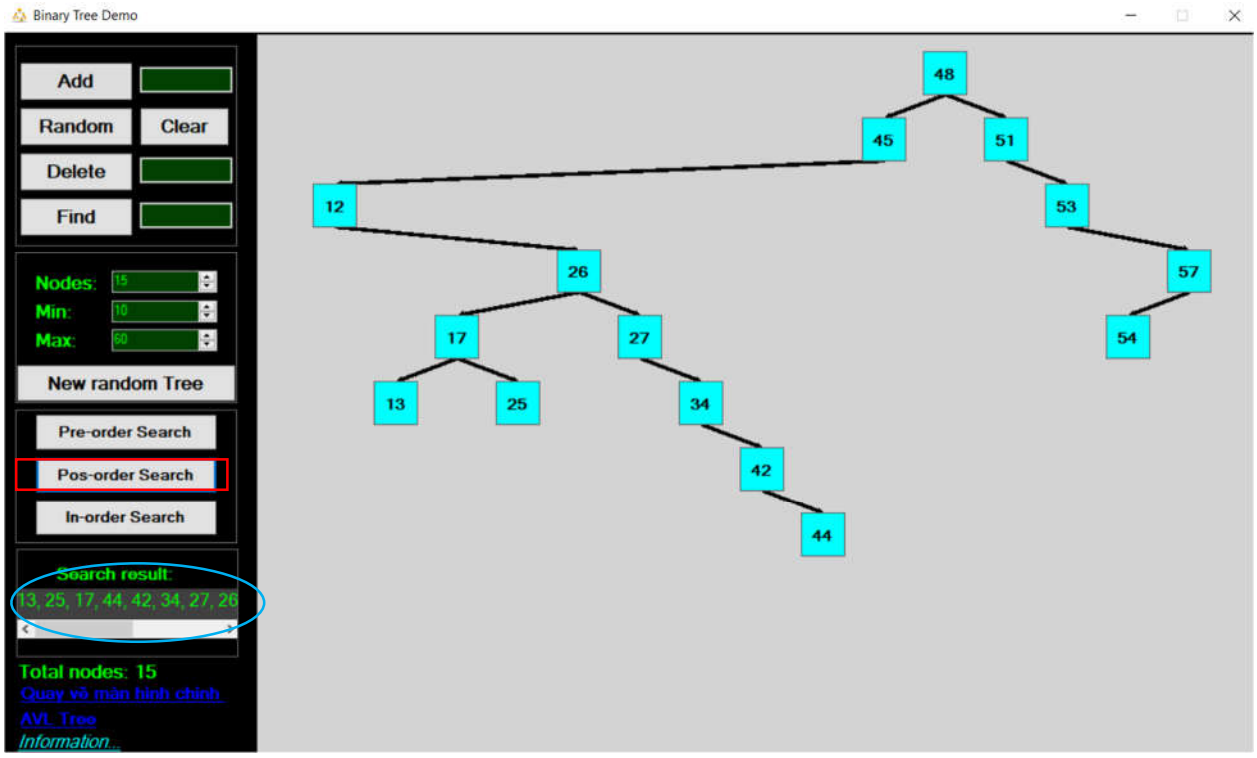
Hình 25– Tìm node 43 bằng nút Find



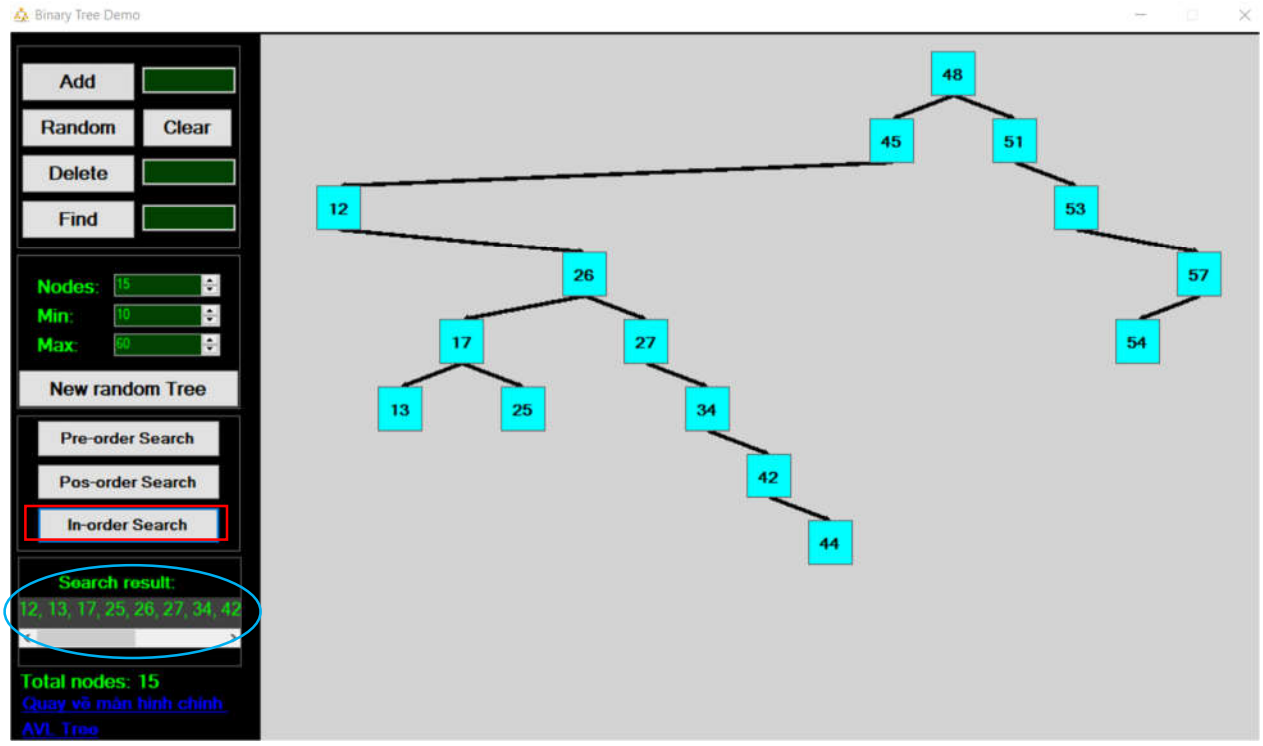
Hình 26– Chọn khoảng giá trị của node và số lượng node cần tạo ngẫu nhiên



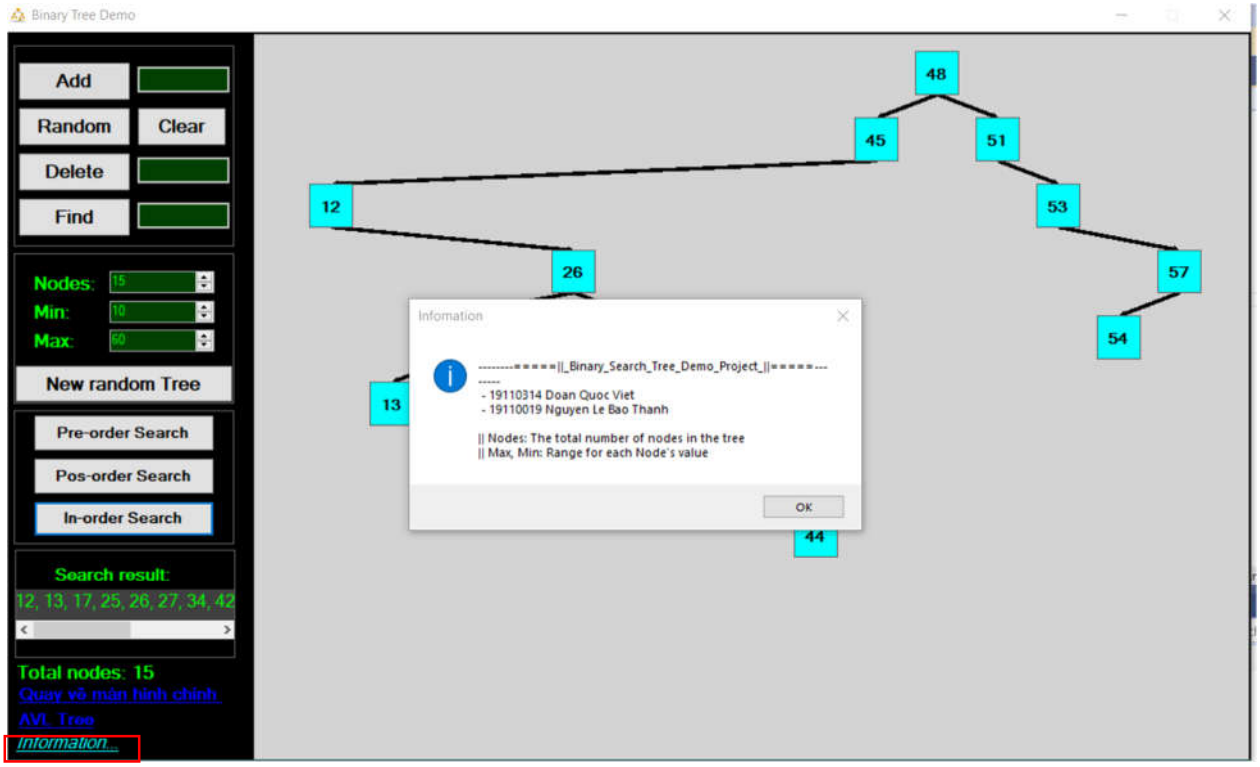
Hình 27– Duyệt trước (Pre-order Search)



Hình 28– Duyệt sau (Post-order Search)



Hình 29– Duyệt sau (In-order Search)



Hình 30 – Hiển thị thông tin ứng dụng (Information)

CHƯƠNG 4: TÌM HIỂU AVL TREE (CÂY CÂN BẰNG)

4.1. Cây cân bằng là gì?

Cây AVL (viết tắt của tên các nhà phát minh Adelson, Velski và Landis) là cây tìm kiếm nhị phân có độ cân bằng cao. Cây AVL kiểm tra độ cao của các cây con bên trái và cây con bên phải và bảo đảm rằng hiệu số giữa chúng là không lớn hơn 1. Hiệu số này được gọi là Balance Factor (Nhân tố cân bằng).

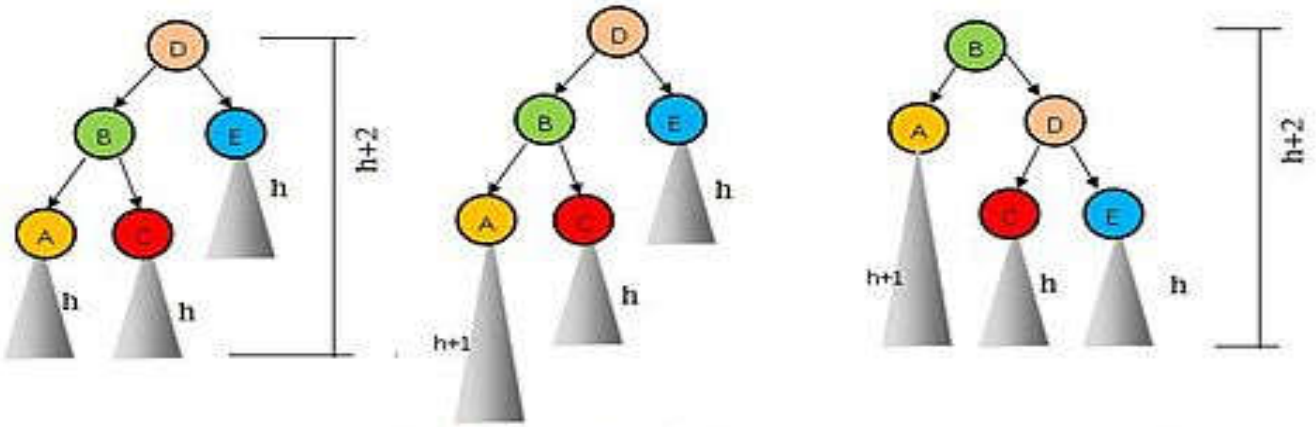
balance(tree)= **height**(tree.right)-**height**(tree.left)

4.2. Các trường hợp mất cân bằng (sau khi insert)

Nếu với mọi đỉnh u của T ta có **balance**(u)= 0 thì T được gọi là cây cân bằng hoàn toàn

Nếu balance(T)< 0, nghĩa là cây con trái cao hơn cây con phải T được gọi là cây lệch trái.

4.2.1. Trường hợp LL



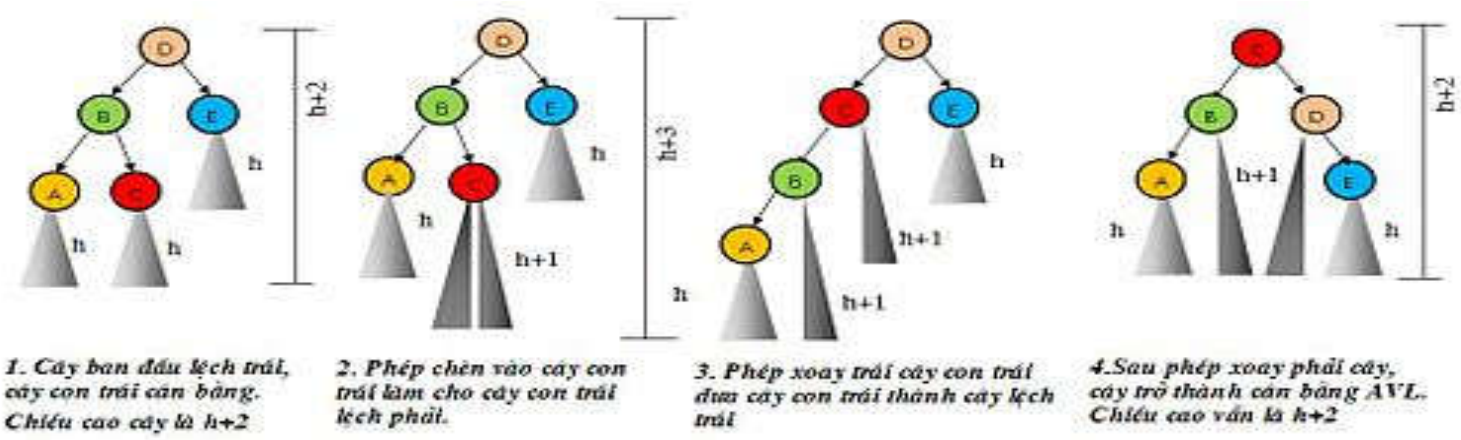
1. Cây lệch trái trước khi chèn . Chiều cao cây là h+2,

2. Chèn một phần tử vào cây con trái của cây con trái làm cho cây mất cân bằng AVL

3. Sau phép quay phải cây trở thanh cân bằng, chiều cao cây vẫn là h+2

Hình 31– Trường hợp LL

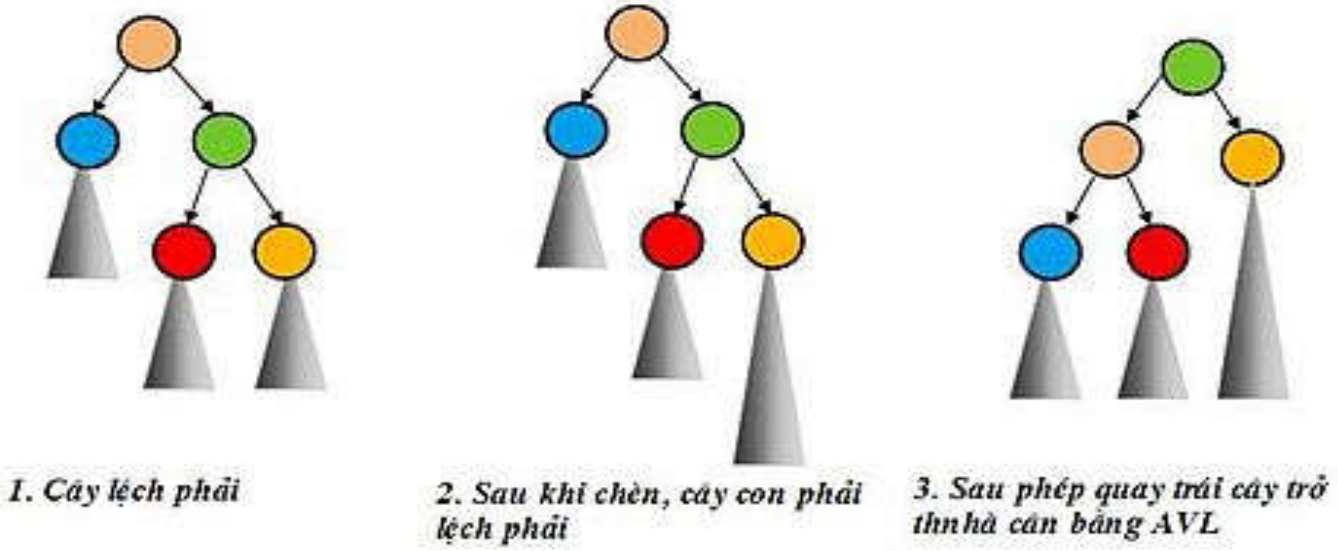
4.2.2. Trường hợp LR



Hình 32– Trường hợp LR

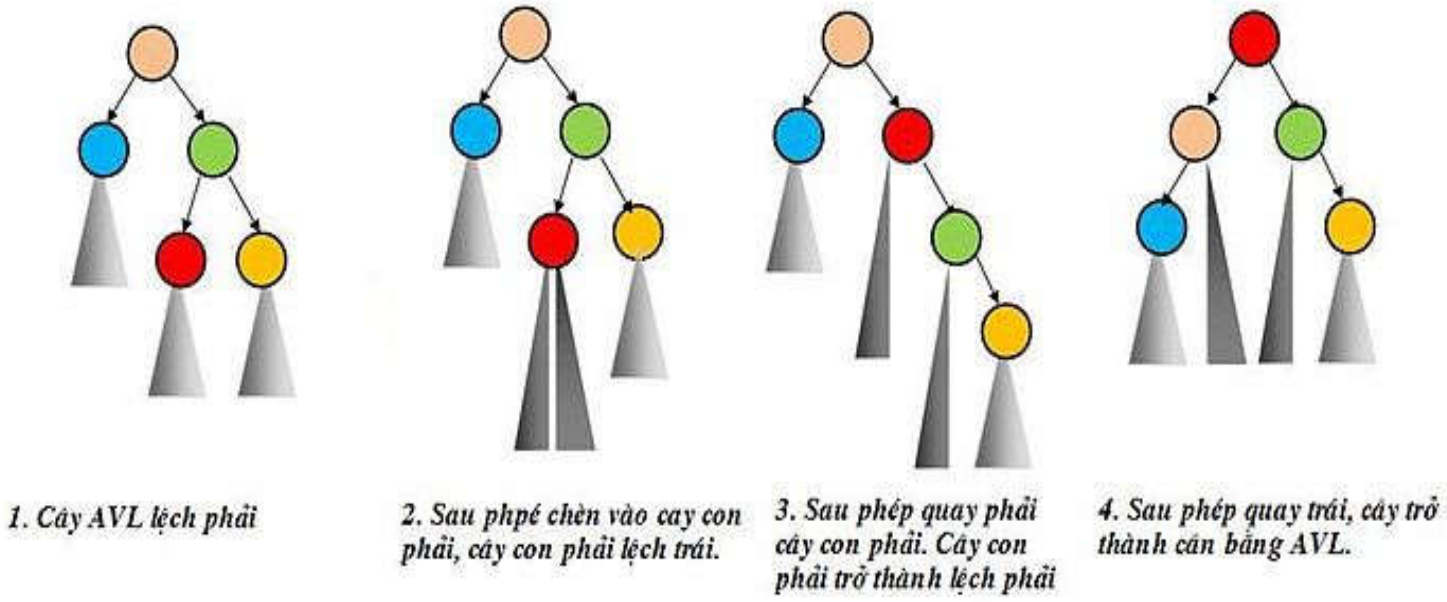
Nếu balance(T) > 0, nghĩa là cây con phải cao hơn cây con trái T được gọi là cây lệch phải

4.2.3. Trường hợp RR



Hình 33– Trường hợp RR

4.2.4. Trường hợp RL



Hình 34– Trường hợp RL

4.3. Ứng dụng

- Cây AVL chủ yếu được sử dụng cho các loại bộ và từ điển trong bộ nhớ.
- Cây AVL cũng được sử dụng rộng rãi trong các ứng dụng cơ sở dữ liệu, trong đó việc chèn và xóa ít hơn nhưng thường xuyên phải tra cứu dữ liệu.
- Nó được sử dụng trong các ứng dụng yêu cầu cải tiến tìm kiếm ngoài các ứng dụng cơ sở dữ liệu.

4.4. Ưu nhược điểm

4.4.1. Ưu điểm

- Chiều cao của cây AVL luôn cân đối. Chiều cao không bao giờ phát triển vượt quá log N, trong đó N là tổng số nút trong cây.
- Nó cho độ phức tạp về thời gian tìm kiếm tốt hơn khi so sánh với cây Tìm kiếm nhị phân đơn giản.
- Cây AVL có khả năng tự cân bằng.

4.4.2. Nhược điểm

- Cây AVL rất khó thực hiện.
- Ngoài ra, cây AVL có hệ số không đổi cao đối với một số hoạt động. Ví dụ: tái cấu trúc là một hoạt động tốn kém và cây AVL có thể phải tự cân bằng lại \ log_2 nlog 2 N trong trường hợp xấu nhất trong quá trình loại bỏ một nút.
- Hầu hết các triển khai STL của các vùng chứa kết hợp có thứ tự (bộ, nhiều tập, bản đồ và nhiều bản đồ) sử dụng cây đỏ-đen thay vì cây AVL. Không giống như cây AVL, cây đỏ-đen chỉ yêu cầu một lần tái cấu trúc để loại bỏ hoặc chèn thêm.

CHƯƠNG 5: MINH HỌA THUẬT TOÁN AVL TREE (CÂY CÂN BẰNG)

5.1. Thiết kế

Hình 35– Mô hình UML của ứng dụng AVL Tree

- Class **AVLTree:**

  Fields:

  ctrThanh, ctrViet, Thanh, Viet: hỗ trợ cho phần hiệu ứng duyệt.

  form: liên kết đến giao diện.

  inOrderResult: kết quả duyệt giữa.

  postOrderResult: kết quả duyệt sau.

  preOrderResult: kết quả duyệt trước.

Root: nút gốc.

Method:

delay: hiệu ứng độ trễ.

deleteNode: xóa node.

find: tìm kiếm node.

getBalance: cân bằng lại cây.

height: chiều cao của cây.

inOrder: duyệt giữa.

inOrderClear, InOrderHelper: trang trí và hiệu ứng duyệt giữa.

PreOrder: duyệt trước.

PreOrderClear, PreOrderHelper: trang trí và hiệu ứng duyệt trước.

PostOrder: duyệt giữa.

PostOrderClear, PostOrderHelper: trang trí và hiệu ứng duyệt sau.

insert: chèn node.

insertHelper: trang trí và hiệu ứng chèn.

leftRotate: xoay trái.

rightRotate: xoay phải.

max: giá trị node lớn nhất.

minValueNode: giá trị node nhỏ nhất.

- Class **Node:**

Fields:

height: chiều cao của cây.

idThanh, idViet: kiểu string để ghi kết quả.

key: giá trị của node.

left: node bên trái.

right: node bên phải.

Method:

NODE: gán giá trị node, cập nhật height.

- Class **Extra:**

  Fields:

  x1, x2, y1, y2: hỗ trợ cho phần chính AVLTree. .

  Method:

  Extra: hỗ trợ cho phần chính AVLTree.

- Class **Extra:**

  Fields:

  x, y: hỗ trợ cho phần chính AVLTree.

  brush: hiệu ứng màu

  value: giá trị

  Method:

  Value: Hỗ trợ cho phần chính AVLTree.

5.2. Code

5.2.1. AVLTree.cs

```
public class AVLTree
{
    // khai báo các biến
    public Node root;
    public string inOrderResult;
    public string preOrderResult;
    public string postOrderResult;
    // lấy chiều cao của cây
    public Dictionary<string, Value> Viet = new Dictionary<string, Value>();
    public Dictionary<string, Extra> Thanh = new Dictionary<string, Extra>();

    Form1 form;
    int ctrThanh = 1;
    int ctrViet = 1;
```

Hình 36– Khai báo các biến cần thiết cho ứng dụng AVL Tree

```
int height(Node N) // chiều cao của cây
{
    if (N == null)
        return 0;
    return N.height;
}

// Hàm tìm số lớn nhất trong 2 số
6 references
int max(int a, int b)
{
    return (a > b) ? a : b;
}

// xoay phải cây con từ y
6 references
Node rightRotate(Node y)
{
    Node x = y.left;
    Node T2 = x.right;

    x.right = y;
    y.left = T2;

    // cập nhật chiều cao
    y.height = max(height(y.left), height(y.right)) + 1;
    x.height = max(height(x.left), height(x.right)) + 1;

    // trả về root mới
    return x;
}
```

Hình 37– Các hàm height (chiều cao của cây), hàm max (số lớn nhất trong 2 số) và rightRotate(xoay phải cây)

```
// xoay phải cây con từ y
6 references
Node rightRotate(Node y)
{
    Node x = y.left;
    Node T2 = x.right;

    x.right = y;
    y.left = T2;

    // cập nhật chiều cao
    y.height = max(height(y.left), height(y.right)) + 1;
    x.height = max(height(x.left), height(x.right)) + 1;

    // trả về root mới
    return x;
}

// xoay trái cây con từ x
6 references
Node leftRotate(Node x)
{
    //y= root mới
    //T2= gốc mới bên trái
    Node y = x.right;
    Node T2 = y.left;

    // Thực hiện xoay
    y.left = x;
    x.right = T2;

    // cập nhật chiều cao
    x.height = max(height(x.left), height(x.right)) + 1;
    y.height = max(height(y.left), height(y.right)) + 1;

    // Return new root
    return y;
}
```

Hình 38– Các hàm xoay

```
// Lấy hệ số cân bằng của nút N
30 references
public void delay() //hiệu ứng độ trễ
{
    form.pictureBox1.Invalidate();
    Application.DoEvents();
    Thread.Sleep(300);

}
6 references
int getBalance(Node N) // điều chỉnh cân bằng lại
{
    if (N == null)
        return 0;
    return height(N.left) - height(N.right);
}
```

Hình 39– Các hàm cân bằng và hiệu ứng

```
public Node InsertHelper(Node node, int key) // chèn node
{
    return insert(node, key, 1, 1, 34);
}
3 references
public Node insert(Node node, int key, int kali, int bagi, int tinggi) // các status diễn biến khi chèn
{
    if (node == null)
    {
        form.Status.Text = "Current is null";
        delay();
        form.Status.Text = "Inserting node";
        delay();

        Viet.Add("Viet" + ctrViet, new Value(484 / bagi * kali + 1, tinggi, key.ToString()));
        ctrViet++;
        form.pictureBox1.Invalidate();
        return (new Node(key, "Viet" + (ctrViet - 1)));

    }

    if (key < node.key)
    {
        form.Status.Text = key + " is lesser than " + node.key;
        Viet[node.idViet].brush = new SolidBrush(Color.Yellow);
        delay();
        form.Status.Text = " Current go to left";
        Viet[node.idViet].brush = new SolidBrush(Color.Black);
        delay();

        node.left = insert(node.left, key, (kali * 2) - 1, bagi * 2, tinggi + 100);
    }
    else if (key > node.key)
    {
        form.Status.Text = key + " is greater than " + node.key;
        Viet[node.idViet].brush = new SolidBrush(Color.Yellow);
        delay();
        form.Status.Text = " Current go to right";
        Viet[node.idViet].brush = new SolidBrush(Color.Black);
        delay();
        node.right = insert(node.right, key, (kali * 2) + 1, bagi * 2, tinggi + 100);
    }
    else
        return node;
```

Hình 40– Hàm chèn node và các hàm thể hiện diễn biến khi chèn

```
/* Cập nhật chiều cao của nút tổ tiên này */
node.height = 1 + max(height(node.left),
                      height(node.right));

/* Lấy hệ số cân bằng của tổ tiên này
node to check whether this node became
Wunbalanced */
int balance = getBalance(node);

// Nếu nút này trở nên không cân bằng, thì có 4 trường hợp
// Trường hợp Left Left
if (balance > 1 && key < node.left.key)
{
    form.Status.Text = "Right Rotate on " + node.key;
    delay();
    return rightRotate(node);
}

// Trường hợp Right Right
if (balance < -1 && key > node.right.key)
{
    form.Status.Text = "Right Rotate on " + node.key;
    delay();
    return leftRotate(node);
}

// Trường hợp Left Right
if (balance > 1 && key > node.left.key)
{
    form.Status.Text = "Double right Rotate on " + node.key;
    delay();
    node.left = leftRotate(node.left);
    return rightRotate(node);
}

// Trường hợp Right Left
if (balance < -1 && key < node.right.key)
{
    form.Status.Text = "Double left Rotate on " + node.key;
    delay();
    node.right = rightRotate(node.right);
    return leftRotate(node);
}

/* trả về con trỏ nút (không thay đổi) */
return node;
```

Hình 41– Các trường hợp mất cân bằng

```
/* Đưa ra một cây tìm kiếm nhị phân không rỗng, trả về
nút có giá trị khóa tối thiểu được tìm thấy trong cây đó.
Lưu ý rằng toàn bộ cây không cần phải
đã tìm kiếm. */
1 reference
Node minValueNode(Node node)
{
    Node current = node;

    /* vòng xuống để tìm lá ngoài cùng bên trái */
    while (current.left != null)
        current = current.left;

    return current;
}

4 references
public Node deleteNode(Node root, int key)
{
    // nếu root là rỗng thì
    if (root == null)
    {
        form.Status.Text = "Deleted key not found in tree";
        delay();
        return root;
    }

    // Nếu khóa cần xóa nhỏ hơn khóa của gốc, thì nó nằm ở cây con bên trái
    if (key < root.key)
    {
        form.Status.Text = key + " is lesser than " + root.key;
        Viet[root.idViet].brush = new SolidBrush(Color.Yellow);
        delay();
        form.Status.Text = " Current go to left";
        Viet[root.idViet].brush = new SolidBrush(Color.Black);
        delay();

        root.left = deleteNode(root.left, key);
    }

    // Nếu khóa cần xóa lớn hơn khóa của gốc, thì nó nằm trong cây con bên phải
    else if (key > root.key)
    {
        form.Status.Text = key + " is greater than " + root.key;
        Viet[root.idViet].brush = new SolidBrush(Color.Yellow);
        delay();
        form.Status.Text = " Current go to right";
        Viet[root.idViet].brush = new SolidBrush(Color.Black);
        delay();

        root.right = deleteNode(root.right, key);
    }
```

Hình 42– Hàm tìm node nhỏ nhất và các trường hợp xóa node

```
// nếu khóa giống với khóa của gốc, thì đây là nút sẽ bị xóa
else
{
    form.Status.Text = "Node is found";
    Viet[root.idViet].brush = new SolidBrush(Color.Yellow);
    delay();

    // nút chỉ có một nút con hoặc không có nút con
    if ((root.left == null) || (root.right == null))
    {
        Node temp = null;
        if (temp == root.left)
            temp = root.right;
        else
            temp = root.left;

        // Trường hợp không có con
        if (temp == null)
        {
            temp = root;
            root = null;
        }
        else // Trường hợp chỉ có một con
            root = temp;
    }
    else
    {

        // nút có hai nút con: Lấy nút kế tiếp nhỏ hơn (nhỏ nhất trong cây con bên phải)
        Node temp = minValueNode(root.right);

        // Sao chép dữ liệu duyệt giữa của successor vào nút này
        root.key = temp.key;

        // Xóa duyệt giữa của successor
        root.right = deleteNode(root.right, temp.key);
    }

}

// Nếu cây chỉ có một nút thì trả về
if (root == null)
    return root;
```

Hình 43– Các trường hợp xóa node (tiếp theo)

```
// Cập nhật chiều cao của node hiện tại
root.height = max(height(root.left), height(root.right)) + 1;

// Kiểm tra xem node có cân bằng không, nếu không thì cân bằng lại
int balance = getBalance(root);

// Nếu node không cân bằng sẽ có 4 trường hợp
// Trường hợp Left Left
if (balance > 1 && getBalance(root.left) >= 0)
{
    form.Status.Text = "Right Rotate on " + root.key;
    delay();
    return rightRotate(root);
}
// Trường hợp Left Right
if (balance > 1 && getBalance(root.left) < 0)
{

    form.Status.Text = "Double Right Rotate on " + root.key;
    delay();

    root.left = leftRotate(root.left);
    return rightRotate(root);
}

// Trường hợp Right Right
if (balance < -1 && getBalance(root.right) <= 0)
{
    form.Status.Text = "Left Rotate on " + root.key;
    delay();
    return leftRotate(root);
}

// Trường hợp Right Left
if (balance < -1 && getBalance(root.right) > 0)
{
    form.Status.Text = "Double Rotate on " + root.key;
    delay();

    root.right = rightRotate(root.right);
    return leftRotate(root);
}

return root;
```

Hình 44– Các trường hợp mất cân bằng (tiếp theo)

```
public void find(int value) // Hàm tìm kiếm node
{
    if (root == null) return;
    Node current = root;
    while (current != null)
    {
        if (value < current.key)
        {
            form.Status.Text = value + " is lesser than " + current.key;
            Viet[current.idViet].brush = new SolidBrush(Color.Yellow);
            delay();
            form.Status.Text = " Current go to left";
            Viet[current.idViet].brush = new SolidBrush(Color.Black);
            delay();
            current = current.left;
        }
        else if (value > current.key)
        {
            form.Status.Text = value + " is greater than " + current.key;
            Viet[current.idViet].brush = new SolidBrush(Color.Yellow);
            delay();
            form.Status.Text = " Current go to right";
            Viet[current.idViet].brush = new SolidBrush(Color.Black);
            current = current.right;
            delay();
        }
        else
        {
            form.Status.Text = value + " exist in tree";
            Viet[current.idViet].brush = new SolidBrush(Color.Yellow);
            delay();
            Viet[current.idViet].brush = new SolidBrush(Color.Black);
            form.mbox(value + " has been found in tree");
            delay();
            return;
        }

    }
    form.Status.Text = value + "Value not found";
    form.mbox(value + " doesn't exist in tree");
    delay();

}
```

Hình 45– Hàm tìm kiếm Node (Find)

```
public void inOrder(Node root)
{
    if (root == null) return;
    inOrder(root.left);
    Viet[root.idViet].brush = new SolidBrush(Color.Yellow);
    delay();
    inOrderResult += root.key + " ";
    inOrder(root.right);

}
4 references
public void preOrder(Node root)
{
    if (root == null) return;
    Viet[root.idViet].brush = new SolidBrush(Color.Yellow);
    delay();
    preOrderResult += root.key + " ";
    preOrder(root.left);
    preOrder(root.right);
}
3 references
public void postOrder(Node root)
{
    if (root == null) return;
    postOrder(root.left);
    postOrder(root.right);
    Viet[root.idViet].brush = new SolidBrush(Color.Yellow);
    delay();
    postOrderResult += root.key + " ";
}
```

Hình 46 – Các hàm duyệt giữa, duyệt trước, duyệt sau

```
// PHAN TRANG TRI VA HIỆU UNG
5 references
public void inOrderHelper()
{
    Viet.Clear();
    Thanh.Clear();
    ctrViet = 1;
    ctrThanh = 1;
    inOrderClear(ref this.root, 1, 1, 50);
}
6 references
public void inOrderClear(ref Node root, int kali, int bagi, int tinggi)
{
    if (root == null) return;
    inOrderClear(ref root.left, (kali * 2) - 1, bagi * 2, tinggi + 100);

    Viet.Add("Viet" + ctrViet, new Value(484 / bagi * kali + 1, tinggi, root.key.ToString()));
    root.idViet = "Viet" + ctrViet;
    ctrViet++;
    form.pictureBox1.Invalidate();
    Application.DoEvents();
    if (root.left != null)
    {
        Thanh.Add("Thanh" + ctrThanh, new Extra((484 / bagi * kali + 1) + 25, tinggi + 30, (484 / (bagi * 2)) * ((kali * 2) - 1) + 25, tinggi + 100));
        ctrThanh++;
        form.pictureBox1.Invalidate();
    }

    if (root.right != null)
    {
        Thanh.Add("Thanh" + ctrThanh, new Extra((484 / bagi * kali + 1) + 25, tinggi + 30, (484 / (bagi * 2)) * ((kali * 2) + 1) + 25, tinggi + 100));
        ctrThanh++;
        form.pictureBox1.Invalidate();
    }
    inOrderClear(ref root.right, (kali * 2) + 1, bagi * 2, tinggi + 100);
}
0 references
public void preOrderHelper()
{
    Viet.Clear();
    Thanh.Clear();
    ctrViet = 1;
    ctrThanh = 1;
    preOrderClear(ref this.root, 1, 1, 50);
}
```

Hình 47– Phần trang trí và hiệu ứng duyệt

```
public void preOrderClear(ref Node root, int kali, int bagi, int tinggi)
{
    if (root == null) return;
    inOrderClear(ref root.left, (kali * 2) - 1, bagi * 2, tinggi + 100);

    Viet.Add("Viet" + ctrViet, new Value(484 / bagi * kali + 1, tinggi, root.key.ToString()));
    root.idViet = "Viet" + ctrViet;
    ctrViet++;
    form.pictureBox1.Invalidate();
    Application.DoEvents();
    if (root.left != null)
    {
        Thanh.Add("Thanh" + ctrThanh, new Extra((484 / bagi * kali + 1) + 25, tinggi + 30, (484 / (bagi * 2)) * ((kali * 2) - 1) + 25, tinggi + 100));
        ctrThanh++;
        form.pictureBox1.Invalidate();
    }

    if (root.right != null)
    {
        Thanh.Add("Thanh" + ctrThanh, new Extra((484 / bagi * kali + 1) + 25, tinggi + 30, (484 / (bagi * 2)) * ((kali * 2) + 1) + 25, tinggi + 100));
        ctrThanh++;
        form.pictureBox1.Invalidate();
    }
    inOrderClear(ref root.right, (kali * 2) + 1, bagi * 2, tinggi + 100);
}
0 references
public void postOrderHelper()
{
    Viet.Clear();
    Thanh.Clear();
    ctrViet = 1;
    ctrThanh = 1;
    postOrderClear(ref this.root, 1, 1, 50);
}
2 references
public void postOrderClear(ref Node root, int kali, int bagi, int tinggi)
{
    if (root == null) return;
    inOrderClear(ref root.left, (kali * 2) - 1, bagi * 2, tinggi + 100);

    Viet.Add("Viet" + ctrViet, new Value(484 / bagi * kali + 1, tinggi, root.key.ToString()));
    root.idViet = "Viet" + ctrViet;
    ctrViet++;
    form.pictureBox1.Invalidate();
    Application.DoEvents();
    if (root.left != null)
    {
        Thanh.Add("Thanh" + ctrThanh, new Extra((484 / bagi * kali + 1) + 25, tinggi + 30, (484 / (bagi * 2)) * ((kali * 2) - 1) + 25, tinggi + 100));
        ctrThanh++;
        form.pictureBox1.Invalidate();
    }

    if (root.right != null)
    {
        Thanh.Add("Thanh" + ctrThanh, new Extra((484 / bagi * kali + 1) + 25, tinggi + 30, (484 / (bagi * 2)) * ((kali * 2) + 1) + 25, tinggi + 100));
        ctrThanh++;
        form.pictureBox1.Invalidate();
    }
    postOrderClear(ref root.right, (kali * 2) + 1, bagi * 2, tinggi + 100);
```

Hình 48– Phần trang trí và hiệu ứng duyệt (tiếp theo)

5.2.2. Node.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AVL_Tree
{
    32 references
    public class Node
    {
        public int key, height; // giá trị node, chiều cao cây
        public Node left, right; // node bên trái, bên phải

        // kiểu string truyền vào chuỗi
        public string idViet;
        public string idThanh;

        1 reference
        public Node(int key,string idViet)
        {
            this.key = key;
            this.idViet = idViet;
            height = 1;

        }
    }
```

Hình 49– Node.cs

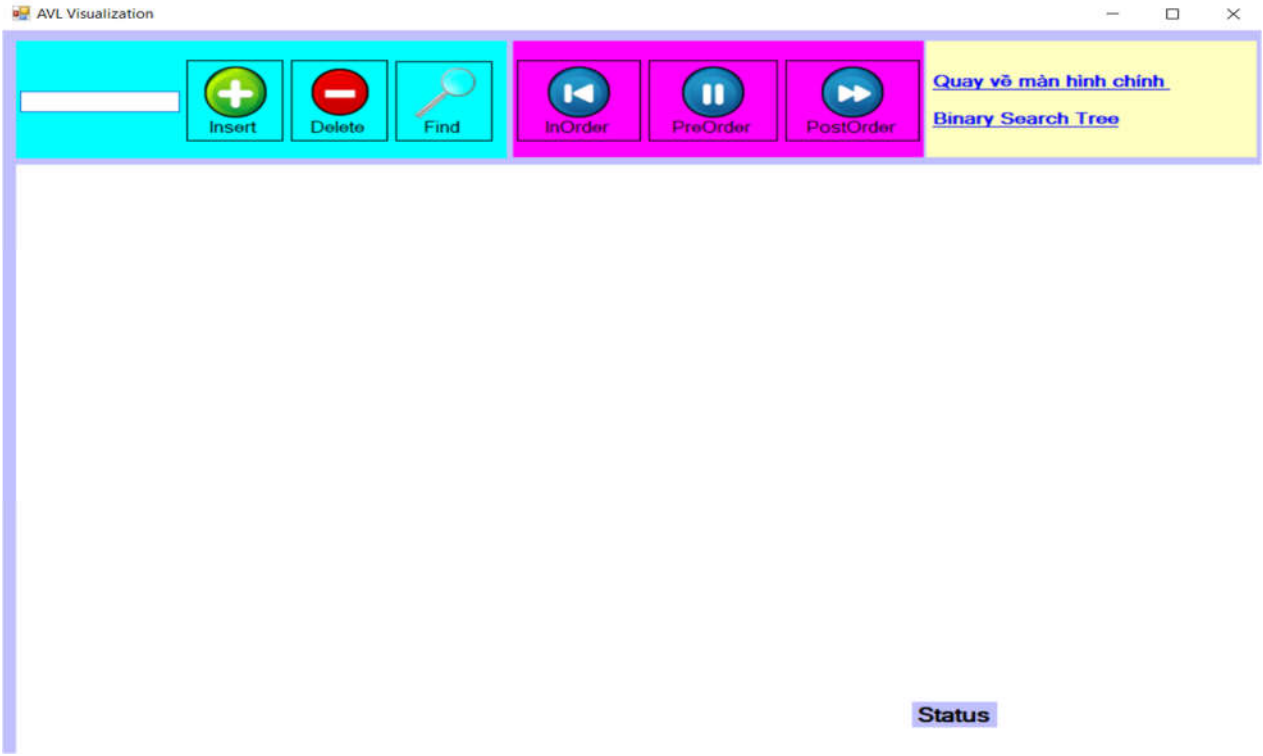
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AVL_Tree
{
    9 references
    public class Extra
    {
        public int x1, x2, y1, y2;
        6 references
        public Extra(int x1, int y1, int x2, int y2) // Hỗ trợ cho AVLTree.cs
        {
            this.x1 = x1;
            this.y1 = y1;
            this.x2 = x2;
            this.y2 = y2;
        }
    }
}
```

Hình 50– Extra.cs

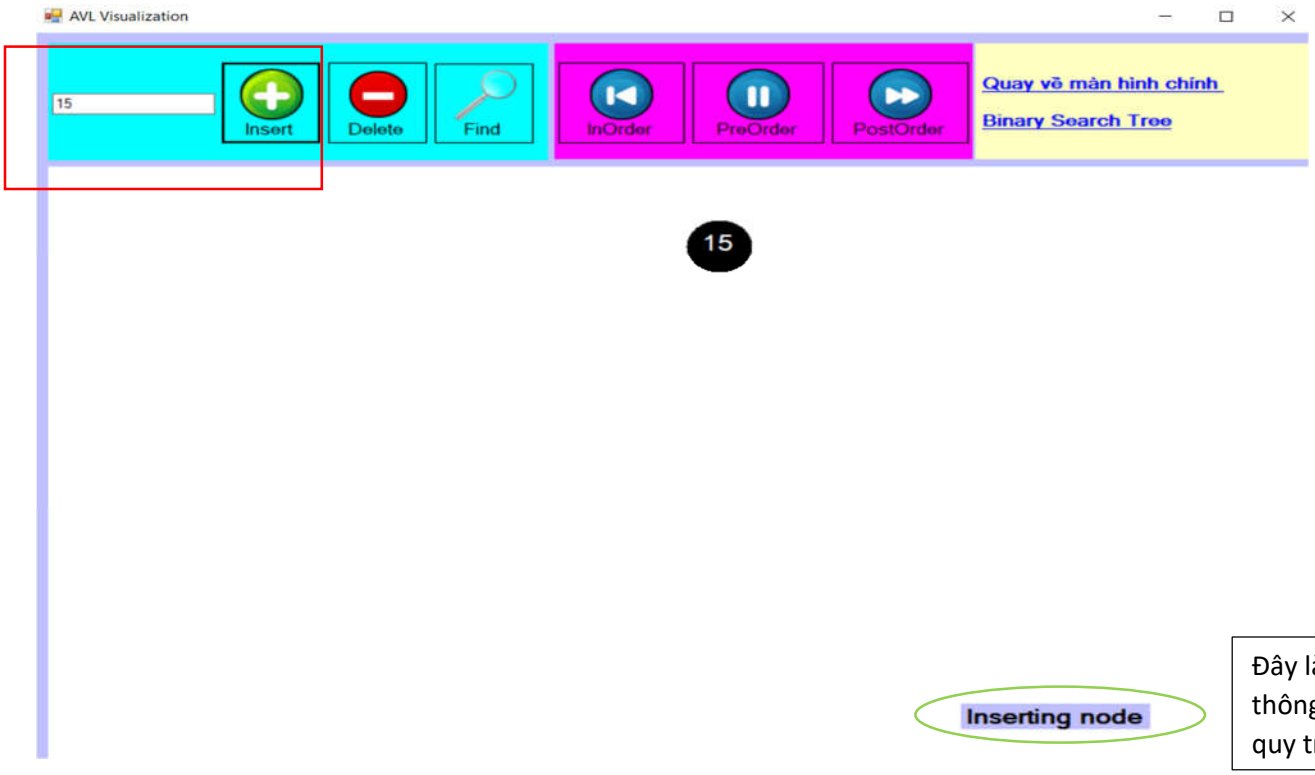
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;
namespace AVL_Tree
{
    7 references
    public class Value
    {
        public int x;
        public int y;
        public string value;
        public Brush brush;
        4 references
        public Value(int x, int y, string value) // gán giá trị và hiệu chỉnh màu đen
        {
            this.x = x;
            this.y = y;
            this.value = value;
            brush = new SolidBrush(Color.Black);
        }
    }

}
```

Hình 51 – Value.cs

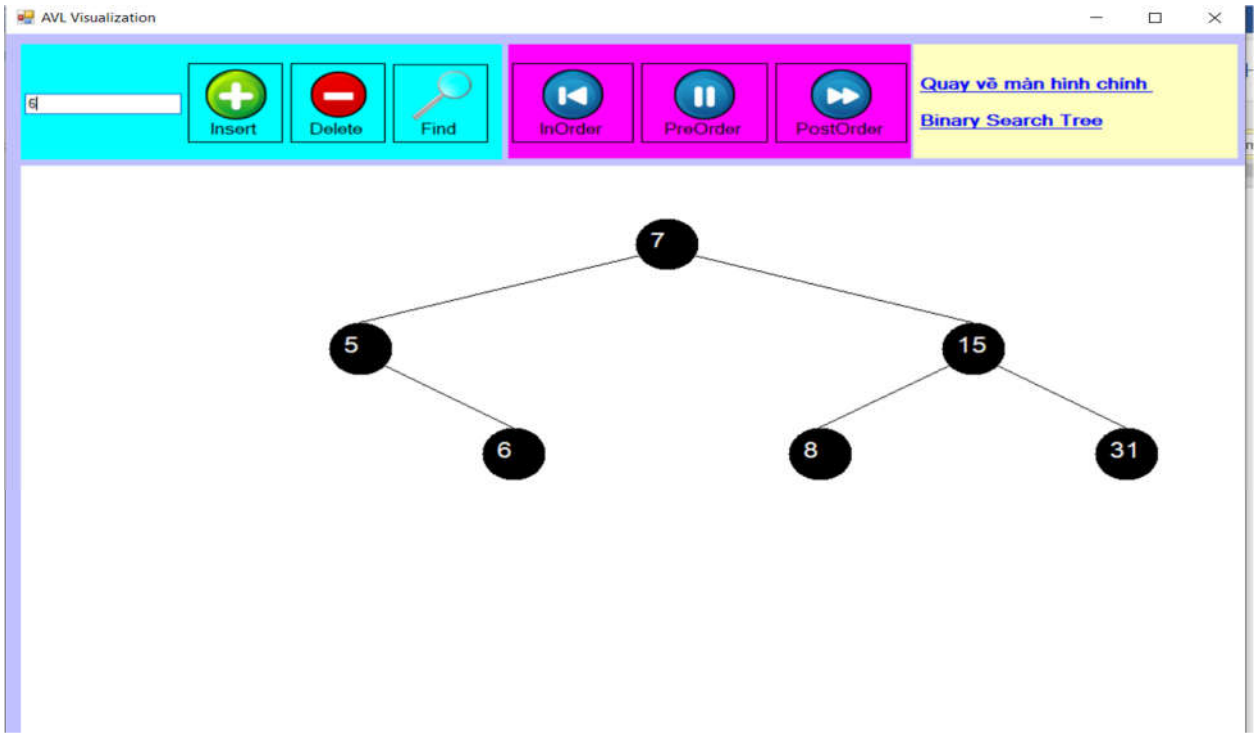
5.3. Giao diện



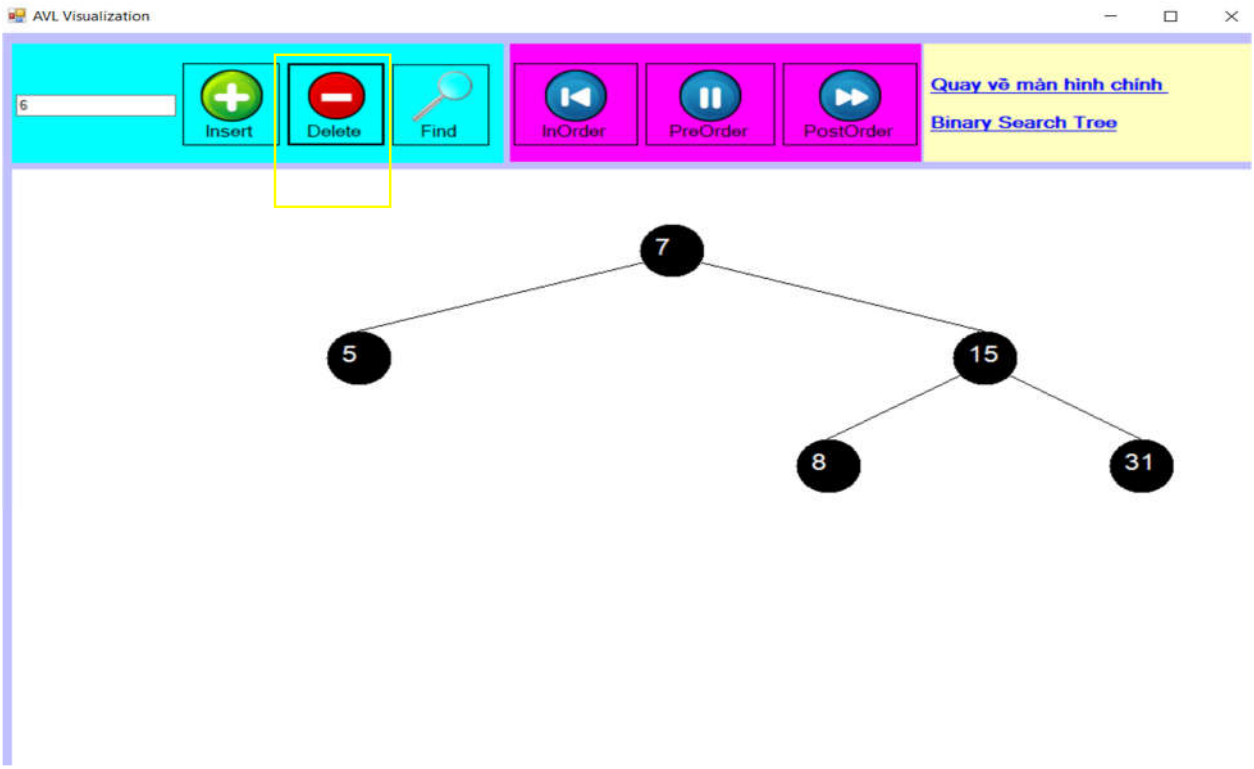
Hình 52– Giao diện chính của ứng dụng demo AVL Tree



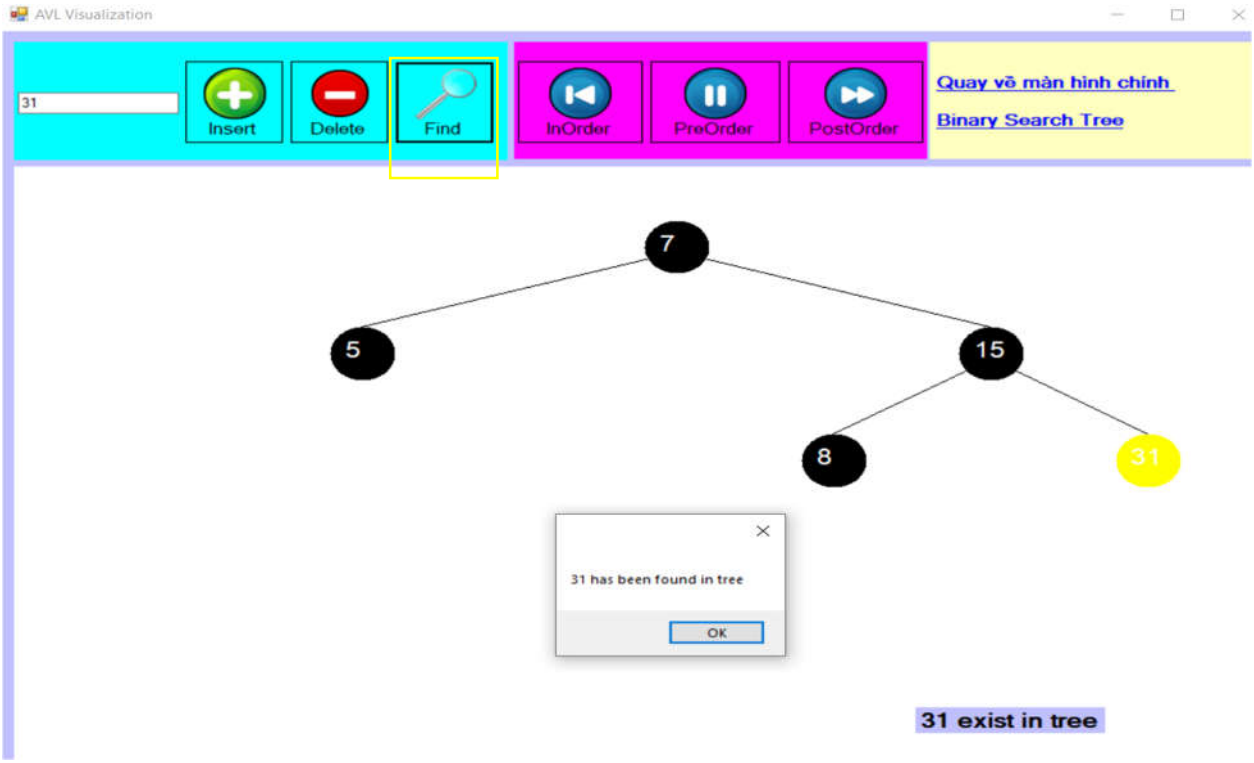
Hình 53– Chèn node 15 bằng nút Insert



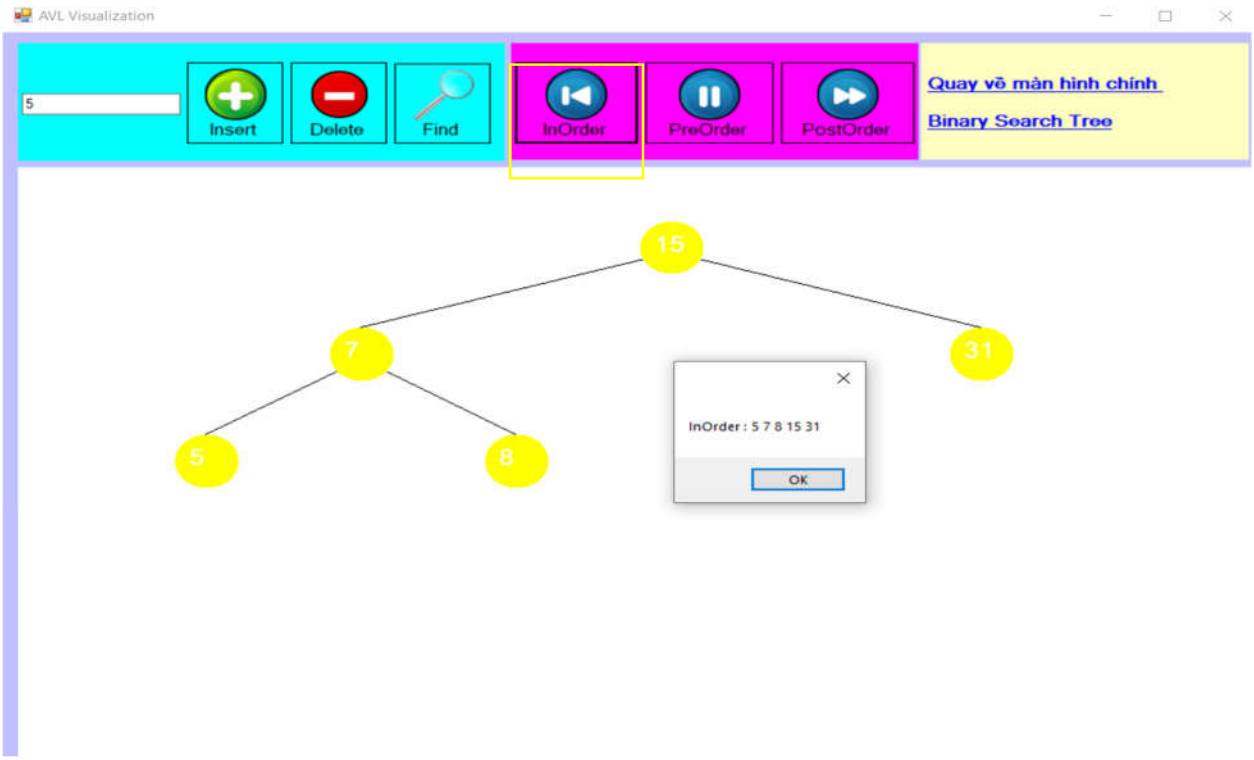
Hình 54– Chèn các node tạo thành 1 cây



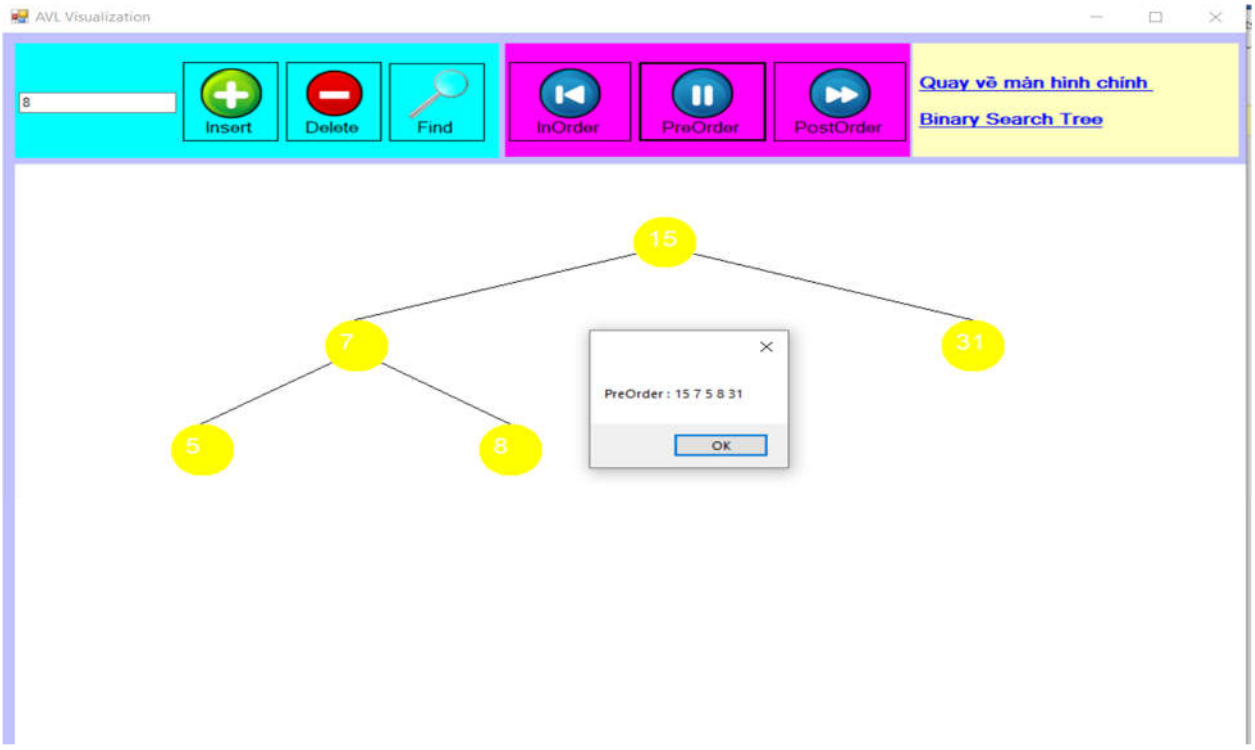
Hình 55– Xóa node 6 bằng nút Delete



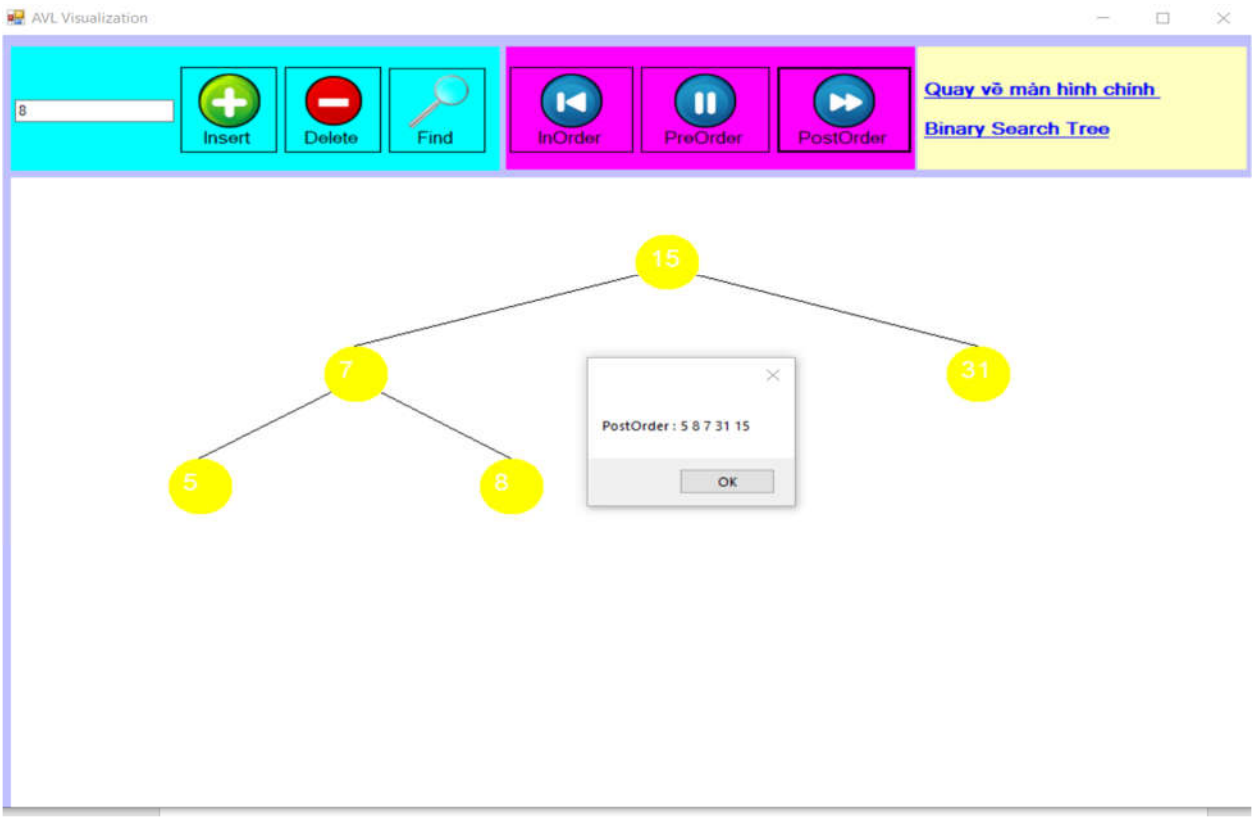
Hình 56– Tìm kiếm node 31 bằng nút Find



Hình 57– Duyệt InOrder



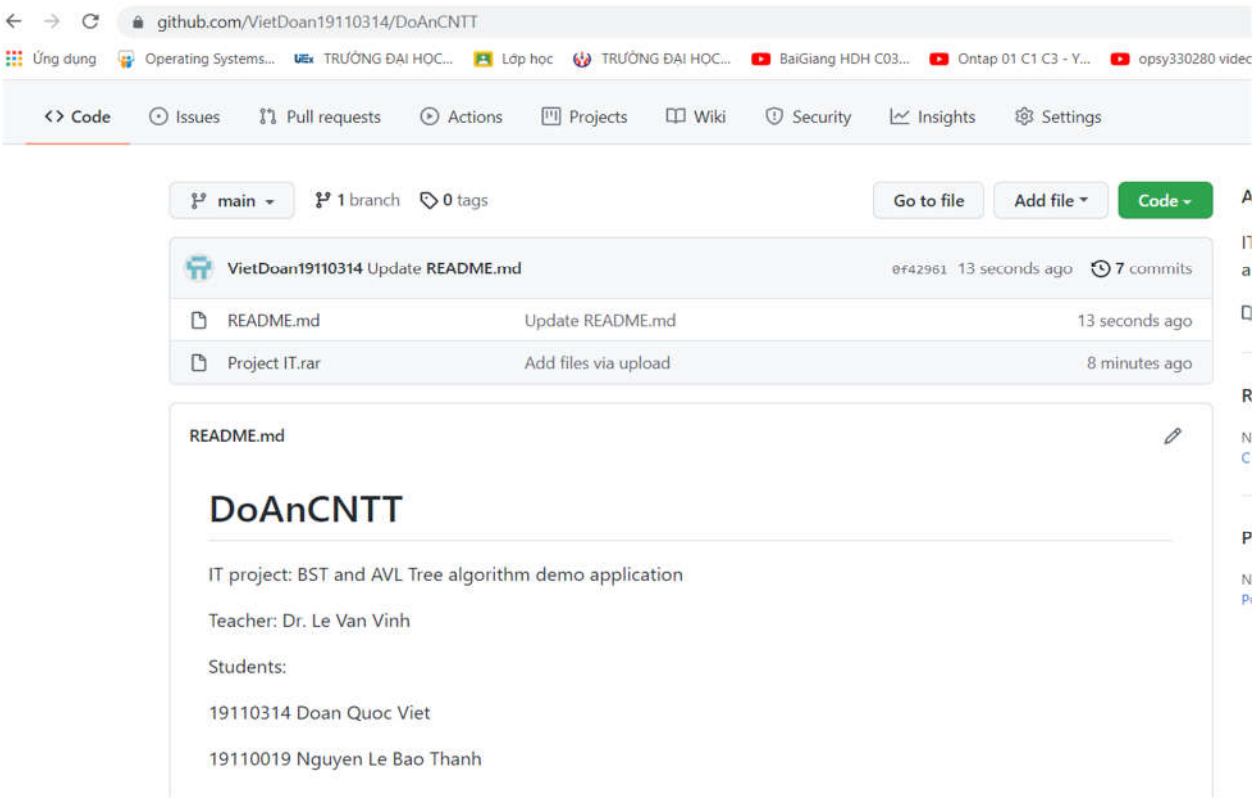
Hình 58– Duyệt PreOrder



Hình 59– Duyệt PostOrder

CHƯƠNG 6: HƯỚNG DẪN SỬ DỤNG ỨNG DỤNG

Bước 1: Vô đường link sau để tải đồ án về:
https://github.com/VietDoan19110314/DoAnCNTT

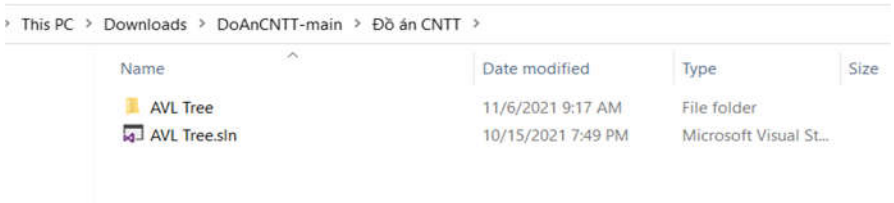


Hình 60– Hướng dẫn tải đồ án

Bước 2: Giải nén file ra, bên trong sẽ xuất hiện tiếp 1 file nén và ta giải nén tiếp

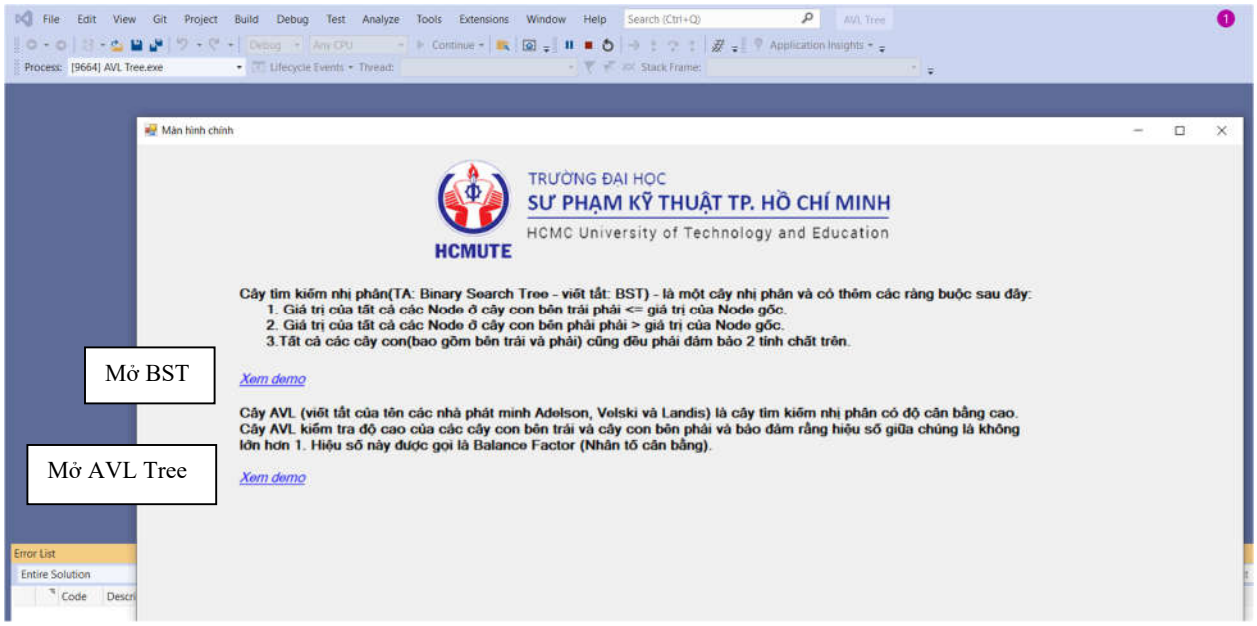
Hình 61– Hướng dẫn giải nén file

Bước 3: Vô file AVL Tree.sln:



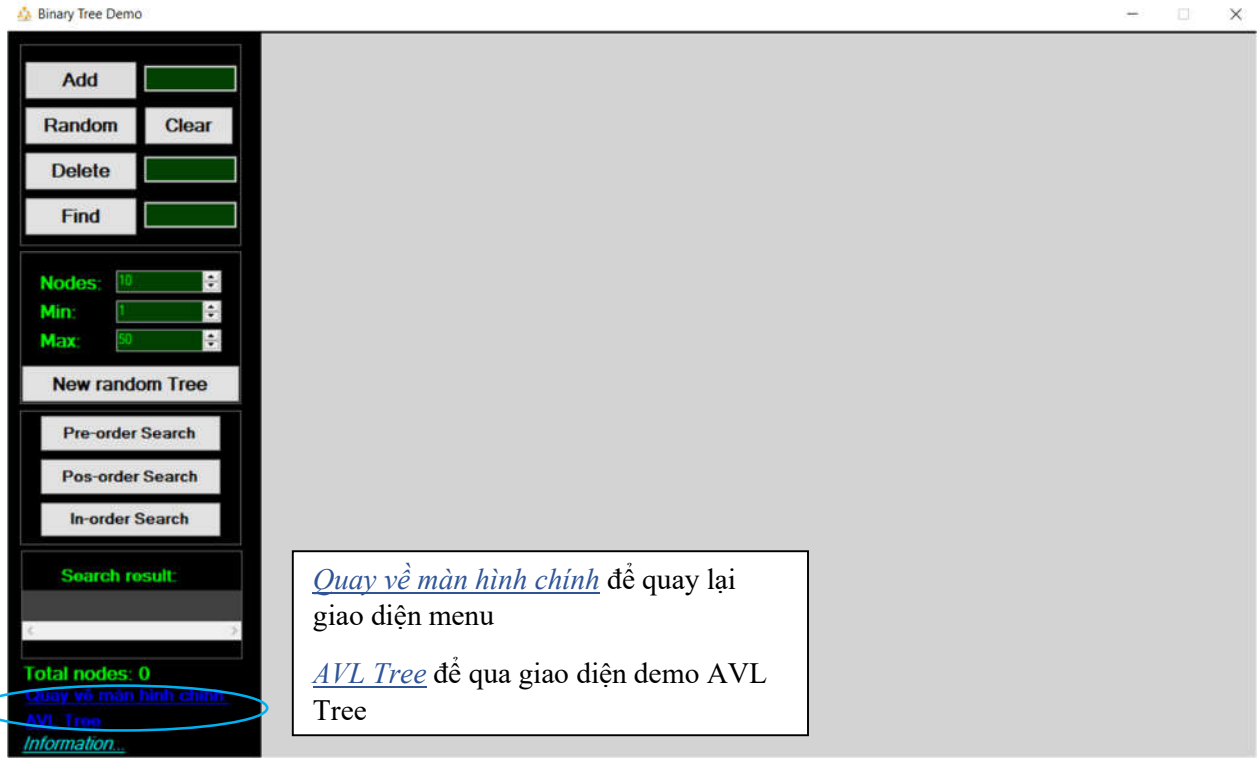
Hình 62– Hướng dẫn mở đồ án

Bước 4: Nhấn Start, giao diện chính sẽ xuất hiện



Hình 63– Giao diện khi khởi động ứng dụng

- Bước 5: Nhấn *Xem demo* ở phía trên để mở giao diện BST



Hình 64– Giao diện chính của BST

Hình 65– Giao diện chính của AVL Tree

CHƯƠNG 7: SO SÁNH GIỮA 2 THUẬT TOÁN BST VÀ AVL TREE

7.1. Giống nhau

- AVL Tree cũng là một BST và có đầy đủ các tính chất như BST.

7.2. Khác nhau

Binary Search tree	AVL tree
Mọi cây tìm kiếm nhị phân là một cây nhị phân vì cả hai cây đều chứa tối đa hai con.	Mọi cây AVL cũng là một cây nhị phân vì cây AVL cũng có hai con tối đa.
Trong BST, không tồn tại một thuật ngữ nào, chẳng hạn như hệ số cân bằng.	Trong cây AVL, mỗi nút chứa một hệ số cân bằng và giá trị của hệ số cân bằng phải là -1, 0 hoặc 1.
Mọi cây Tìm kiếm nhị phân không phải là cây AVL vì BST có thể là một cây cân bằng hoặc không cân bằng.	Mỗi cây AVL là một cây tìm kiếm nhị phân vì cây AVL tuân theo thuộc tính của BST.
Mỗi nút trong cây Tìm kiếm nhị phân bao gồm ba trường, tức là cây con bên trái, giá trị nút và cây con bên phải.	Mỗi nút trong cây AVL bao gồm bốn trường, tức là cây con bên trái, giá trị nút, cây con bên phải và hệ số cân bằng.
Trong trường hợp cây Tìm kiếm nhị phân, nếu chúng ta muốn chèn bất kỳ nút nào vào cây thì chúng ta so sánh giá trị nút với giá trị gốc; nếu giá trị của nút lớn hơn giá trị của nút gốc thì nút đó được chèn vào cây con bên phải, ngược lại nút được chèn vào cây con bên trái. Sau khi nút được chèn, không cần kiểm tra hệ số cân bằng chiều cao để hoàn thành việc chèn.	Trong trường hợp cây AVL, trước tiên, chúng ta sẽ tìm vị trí thích hợp để chèn nút. Khi nút được chèn vào, chúng tôi sẽ tính toán hệ số cân bằng của mỗi nút. Nếu hệ số cân bằng của mỗi nút được thỏa mãn thì việc chèn được hoàn thành. Nếu hệ số cân bằng lớn hơn 1 thì ta cần thực hiện một số phép quay để cây cân bằng.
Trong cây Tìm kiếm nhị phân, chiều cao hoặc độ sâu của cây là O (n) trong đó n là số nút trong cây Tìm kiếm nhị phân.	Trong cây AVL, chiều cao hoặc chiều sâu của cây là O (logn).

Nó đơn giản để thực hiện vì chúng ta phải tuân theo các thuộc tính Tìm kiếm nhị phân để chèn nút.	Nó phức tạp để thực hiện vì trong cây AVL, trước tiên chúng ta phải xây dựng cây AVL, và sau đó chúng ta cần kiểm tra sự cân bằng chiều cao. Nếu chiều cao mất cân đối thì chúng ta cần thực hiện một số thao tác luân phiên để cây cân bằng.
BST không phải là cây cân bằng vì nó không tuân theo khái niệm về hệ số cân bằng.	BST không phải là cây vì nó không tuân theo khái niệm về hệ thống cân bằng. Cây AVL là cây cân bằng chiều cao vì nó tuân theo khái niệm về hệ số cân bằng.
Tìm kiếm không hiệu quả trong BST khi có số lượng lớn các nút có sẵn trong cây vì chiều cao không cân bằng.	Tìm kiếm hiệu quả trong cây AVL ngay cả khi có số lượng lớn các nút trong cây vì chiều cao được cân bằng.

CHƯƠNG 8: TỔNG KẾT

8.1. Ưu điểm

- Đồ án có giao diện đẹp mắt, dễ sử dụng.
- Đồ án có đầy đủ các tính năng cơ bản của thuật toán như thêm, xóa, tìm kiếm, duyệt trước, duyệt giữa, duyệt sau.
- Code sạch dễ đọc.

8.2. Nhược điểm

- Phần demo AVL Tree chưa có phần tạo random 1 cây nên hơi mất thời gian để tạo.

8.3. Ý tưởng phát triển

- Nếu có thời gian và cơ hội chúng em sẽ tiếp tục phát triển các thuật toán này thành cây có mức độ khó hơn như Red-Black Tree, B-Trees…

TÀI LIỆU THAM KHẢO

1. Các tài liệu do thầy TS. Lê Văn Vinh cung cấp
   1.1. Introduction to the Design and Analysis of Algorithms (3rd ed.) [Levitin 2011-10-09]
   1.2. t._cormen_-_introduction_to_algorithms_3rd_edition
2. Giáo trình cấu trúc dữ liệu và giải thuật-Trần Hạnh Nhi
3. Các bài giảng môn Lập trình trên Window
4. Paint Application in C# Visual Studio By Rohit Programming Zone [5 thg 5,2021]
   https://www.youtube.com/watch?v=m7Ohm52TIhw&t=251s
5. Cây tìm kiếm nhị phân [sửa đổi lần cuối vào ngày 17 tháng 5 năm 2021 lúc 08:39]
   https://vi.wikipedia.org/wiki/ Cây_tìm_kiếm_nhị_phân
6. Binary Tree Data Structure [Last Updated : 31 Jul, 2021]
   https://www.geeksforgeeks.org/binary-tree-data-structure/
7. Cây AVL [sửa đổi lần cuối vào ngày 15 tháng 2 năm 2021 lúc 16:58.]
   https://vi.wikipedia.org/wiki/ Cây_AVL
8. Cây (cấu trúc dữ liệu) [sửa đổi lần cuối vào ngày 13 tháng 2 năm 2021 lúc 15:02.]
   https://vi.wikipedia.org/wiki/ Cây_(cấu_trúc_dữ_liệu)
9. Tree (data structure) [last edited on 15 October 2021, at 19:00 (UTC)]
   https://en.wikipedia.org/wiki/Tree_(data_structure)
10. AVL Trees: Rotations, Insertion, Deletion with C++ Example [By Alyssa Walker - Updated October 8, 2021]

https://www.guru99.com/avl-tree.html#11