BÁO CÁO THỰC HÀNH TUẦN 7

24120261 – Đặng Bùi Thế Bảo

I. Cây nhị phân (Binary Tree)

a. Giới thiệu:

- ❖ Cây nhị phân (Binary Tree) là một cấu trúc dữ liệu dạng cây, trong đó mỗi node có tối đa 2 node con, thường gọi là:
  - Node bên trái.
  - Node bên phải.

b. Các hàm tương tác với cây nhị phân:

- ❖ Tạo node mới: ***NODE* createNode(int data);***
  - Tạo ra một node mới cho cây từ giá trị cho trước.

```
NODE* newnode = new NODE();
newnode->key = data;
newnode->p_left = nullptr;
newnode->p_right = nullptr;
return newnode;
```

- ❖ Duyệt cây NLR: ***vector<int> NLR(NODE* pRoot);***
  - Duyệt cây theo chiều sâu (Depth First Search) Pre - order. Bắt đầu từ node root duyệt hết nhánh bên trái rồi tới nhánh bên phải.
  - Dùng đệ quy để duyệt. Dùng vector result để lưu đáp án.

```
vector<int> result;
if(pRoot == nullptr) return result;
result.push_back(pRoot->key);
vector<int> left = NLR(pRoot->p_left);
vector<int> right = NLR(pRoot->p_right);
result.insert(result.end(), left.begin(), left.end());
result.insert(result.end(), right.begin(), right.end());
return result;
```

❖ Duyệt cây LNR: ***vector<int> LNR(NODE* pRoot);***

- Duyệt cây theo chiều sâu (Depth First Search) In - order. Bắt đầu từ node lá bên trái nhất hết nhánh bên trái rồi tới node giữa rồi nhánh bên phải.
- Dùng đệ quy để duyệt. Dùng vector result để lưu.

```
vector<int> result;
    if(pRoot == nullptr) return result;
    vector<int> left = LNR(pRoot->p_left);
    result.insert(result.end(), left.begin(), left.end());
    result.push_back(pRoot->key);
    vector<int> right = LNR(pRoot->p_right);
    result.insert(result.end(), right.begin(), right.end());
    return result;
```

❖ Duyệt cây LRN: ***vector<int> LRN(NODE* pRoot***);

- Duyệt cây theo chiều sâu (Depth First Search) Post - order. Bắt đầu từ node lá bên trái nhất hết nhánh bên trái rồi tới node phải rồi tới node giữa.
- Dùng đệ quy để duyệt. Dùng vector result để lưu.

```
vector<int> result;
        if(pRoot == nullptr) return result;
        vector<int> left = LRN(pRoot -> p_left);
        result.insert(result.end(),left.begin(),left.end());
        vector<int> right = LRN(pRoot -> p_right);
        result.insert(result.end(),right.begin(),right.end());
        result.push_back(pRoot->key);
        return result;
```

❖ Breath First Search: ***vector<vector<int>> LevelOrder(NODE* pRoot);***

- Duyệt cây theo chiều rộng (Breath First Search). Bắt đầu từ node lá duyệt theo từng hàng một cho đến hết.
- Dùng hàng đợi (queue) để duyệt. Lưu bằng mảng vector.

```
vector<NODE*> queue;
    NODE* cur;
    queue.push_back(pRoot);
    while(!queue.empty()){
        int size = queue.size();
        vector<int> temp;
        for(int i = 0; i < size; i++){
            temp.push_back(queue[0]->key);
            cur = queue[0];
            queue.erase(queue.begin());
if(cur->p_left != nullptr) queue.push_back(cur -> p_left);
if(cur->p_right != nullptr) queue.push_back(cur -> p_right);
        }
        result.push_back(temp);
    }
```

❖ Đếm Node: ***int countNode(NODE* pRoot);***

- Đếm tổng số node trong cây bằng cách duyệt cây theo kiểu Pre-order.
- Sử dụng biến đếm để lưu kết quả và dùng đệ quy để duyệt, tại mỗi lần duyệt qua một node biến đếm tự động tăng lên một.

❖ Tính tổng các Node: ***int sumNODE(NODE* pRoot);***

- Tính tổng số node trong cây bằng cách duyệt cây theo kiểu Pre-order.
- Sử dụng biến tổng để lưu kết quả và dùng đệ quy để duyệt, mỗi lần duyệt qua một node cộng giá trị của node đó vào biến tổng.

❖ Vị trí của giá trị: ***int heightNode(NODE* pRoot, int val);***

- Duyệt qua cây để tìm kiếm giá trị cho trước trong cây rồi trả độ sâu của node chứa giá trị đó.

```
if(pRoot == nullptr) return -1;
if(pRoot->key == val) return 0;
int leftHeight = heightNode(pRoot->p_left, val);
int rightHeight = heightNode(pRoot->p_right, val);
if(leftHeight == -1 && rightHeight == -1) return -1;
return max(leftHeight, rightHeight) + 1;
```

❖ Tìm chiều cấp Node: int Level(NODE* pRoot, NODE* p)

- Sử dụng hàng đợi queue để duyệt cây theo chiều rộng, tìm kiếm Node cho trước rồi trả về cấp của Node đó

```
queue.push_back(pRoot);
  while(!queue.empty()){
    int size = queue.size();
    for(int i = 0; i < size; i++){
      if(queue[0] == p) return level;
      cur = queue[0];
      queue.erase(queue.begin());
      if(cur->p_left != nullptr)
        queue.push_back(cur->p_left);
      if(cur->p_right != nullptr)
        queue.push_back(cur->p_right);
    }
    level++;
  }
```

❖ Đếm lá: ***int countLeaf(NODE* pRoot);***

- Dùng đệ quy để duyệt cây bên trái và phải để tìm số lá bên trái và phải rồi cộng lại. Node lá là những node mà hai node con trái, phải của nó đều trỏ về null.

```
if(pRoot == nullptr) return 0;
if(pRoot->p_left == nullptr && pRoot->p_right == nullptr)
  return 1;
return countLeaf(pRoot->p_left) + countLeaf(pRoot->p_right);
```

II. Cây nhị phân tìm kiếm (Binary Search Tree – BST)

a. Giới thiệu:

❖ Cây nhị phân tìm kiếm là một cây nhưng có câu trúc:

- Một node root N, node bên trái nhỏ hơn N nhỏ hơn node bên phải.

b. Các hàm tương tác với cây:

❖ Hàm tìm kiếm: ***NODE* Search(NODE* pRoot, int x);***

- Duyệt cây theo chiều sâu Pre-order để tìm kiếm Node chứa giá trị cho trước.

❖ Hàm chèn: ***void Insert(NODE* &pRoot, int x);***

- Tìm kiếm vị trí cần chèn theo nguyên tắc cây nhị phân tìm kiếm giá trị nhỏ hơn giá trị root đệ quy bên trái ngược lại đệ quy bên phải. Thực hiện thao tác chèn.

❖ Hàm xóa: ***void Remove(NODE* &pRoot, int x);***

- Thực hiện thao tác tìm kiếm để kiếm node chứa giá trị cho trước thực hiện thao tác xóa.
- Với node lá ta thực hiện xóa trực tiếp.

```
NODE* temp = pRoot;
pRoot = pRoot->p_right;
delete temp;
```

- Với node cành khuyết một giá trị bên phải hoặc bên trái ta thực hiện cập nhật liên kết node con với node cha, rồi xóa.

```
NODE* temp = pRoot;
pRoot = pRoot->p_left;
delete temp;
```

- Với node có đủ liên kết trái phải ta thực hiện tìm node lớn nhất bên trái hoặc node nhỏ nhất bên phải để thay thế rồi xóa đi node đó.

```
NODE* temp = pRoot->p_right;
while(temp->p_left != nullptr) temp = temp->p_left;
pRoot->key = temp->key;
Remove(pRoot->p_right, temp->key);
```

❖ Hàm tạo cây từ mảng: ***NODE* createTree(int a[], int n);***

- Ta thực hiện tuần tự các thao tác duyệt mảng để lấy giá trị, rồi thực hiện tìm vị trí thích hợp, thực hiện tạo node mới rồi chèn vào cây.

```
NODE* tree = new NODE();
        for (int i = 1; i<n; i++){
                Insert(tree,a[i]);
        }
        return tree;
```

❖ Hàm xóa cây: ***void removeTree(NODE* &pRoot);***

- Thực hiện duyệt cây và xóa từng node.

```
if(pRoot == nullptr) return;
        removeTree(pRoot->p_left);
        removeTree(pRoot->p_right);
        delete pRoot;
        pRoot = nullptr;
```

❖ Hàm tính chiều cao của cây: ***int Height(NODE* pRoot);***

- Dùng hàng đợi queue để thực hiện duyệt cây theo chiều rộng, (hay duyệt từng hàng). Mỗi khi duyệt xong một hàng thì tăng độ cao lên 1

```
        vector<NODE*> queue;
           queue.push_back(pRoot);
           size_t front = 0;
           int height = 0;

           while (front < queue.size()) {
              int size = queue.size() - front;
              for (int i = 0; i < size; ++i) {
                 NODE* node = queue[front++];
                 if (node->p_left) queue.push_back(node->p_left);
                 if (node->p_right) queue.push_back(node->p_right);
              }
              height++;
           }
```

❖ Hàm đếm các node có giá trị nhỏ hơn giá trị cho trước:
int countLess(NODE* pRoot, int x);

- Thực hiện duyệt cây để tìm những node nhỏ hơn giá trị cho trước và tăng biến đếm.
- Sử dụng đệ quy để thực hiện đếm

```
if(pRoot == nullptr)return 0;
        int count = 0;
        if(x <= pRoot->key) count = countLess(pRoot->p_left, x);
        else {
            count += countLess(pRoot->p_right, x) +
countLess(pRoot->p_left,x) + 1;
        }
        return count;
```

❖ Hàm đếm các node có giá trị lớn hơn giá trị cho trước:
int countGreater(NODE* pRoot, int x);

- Tương tự như đếm node có giá trị nhỏ hơn giá trị cho trước, chỉ thay đổi điều kiện.

```
if(pRoot == nullptr)return 0;
   int count = 0;
   if(x >= pRoot->key) count = countGreater(pRoot->p_right, x);
   else {
       count += countGreater(pRoot->p_left, x) +
countGreater(pRoot->p_right,x) + 1;
   }
   return count;
```

❖ Hàm kiểm tra cây nhị phân: ***bool isBST(NODE* pRoot);***

- Dựa vào tính chất để kiểm tra: node bên trái nhỏ hơn root nhỏ hơn node bên phải.

```
if(pRoot == nullptr) return true;
   if(pRoot->p_left != nullptr && pRoot->p_left->key > pRoot->key)
   return false;
if(pRoot->p_right != nullptr && pRoot->p_right->key < pRoot->key)
return false;
   return isBST(pRoot->p_left) && isBST(pRoot->p_right);
```

❖ Hàm kiểm tra cây nhị phân đầy đủ: ***bool isFullBST(NODE* pRoot);***

- Tính chất cây nhị phân đầy đủ là các node root phải có đủ 2 node trái, phải hoặc khuyết cả hai node.

```
if(pRoot == nullptr) return true;
    if((pRoot->p_left == nullptr && pRoot->p_right != nullptr) ||
(pRoot->p_left != nullptr && pRoot->p_right == nullptr)) return
false;
    return isFullBST(pRoot->p_left) && isFullBST(pRoot->p_right);
```

III. Cây AVL.

a. Giới thiệu: là một cây nhị phân, có tính chất là độ cao của nhánh bên trái và cánh bên phải bằng nhau hoặc chênh lệch một đơn vị.

```
struct NODE{
   int key;
   NODE* p_left;
   NODE* p_right;
   int height;
};
```

b. Các hàm tương tác:

❖ Hàm lấy chiều cao cây: ***int getHeight(NODE* pRoot);***

- Sử dụng để lấy chiều cao của cây.

```
int getHeight(NODE* pRoot) {
   if (pRoot == nullptr) return 0;
   return pRoot->height;
}
```

❖ Hàm cập nhật chiều cao cây:

- Sử dụng để cập nhật lại chiều cao của cây sau khi chèn, hoặc xóa

❖ Hàm xoay trái: ***int updateHeight(NODE* pRoot);***

- Sử dụng để xoay trái tại node mất cân bằng.

```
NODE* temp = pRoot -> p_right;
        NODE* temp2 = temp -> p_left;
        temp->p_left = pRoot;
        pRoot -> p_right = temp2;
        updateHeight(pRoot);
        updateHeight(temp);
        return temp;
```

❖ Hàm xoay phải:

- Sử dụng để xoay trái tại node mất cân bằng.

```
NODE* temp = pRoot -> p_left;
        NODE* temp2 = temp -> p_right;
        temp->p_right = pRoot;
        pRoot -> p_left= temp2;
        updateHeight(pRoot);
        updateHeight(temp);
        return temp;
```

❖ Hàm chèn một node: ***void Insert(NODE* &pRoot, int x);***

- Thực hiện thao tác chèn như cây nhị phân, sau đó cập nhật chiều cao cây và thực hiện thao tác cân bằng cây nếu cây mất cân bằng tạo các trường hợp L-L, L-R, R-L, R-R;

```
updateHeight(pRoot);

    int check = getHeight(pRoot->p_left) - getHeight(pRoot->p_right);
    if(check > 1){
        if(x < pRoot->p_left->key) pRoot = RightRotate(pRoot);
        else if(x>pRoot->p_left->key){
            pRoot->p_left = LeftRotate(pRoot->p_left);
            pRoot = RightRotate(pRoot);
        }
    }
    if(check < -1){
        if(x > pRoot->p_right->key) pRoot = LeftRotate(pRoot);
        else if(x<pRoot->p_right->key){
            pRoot->p_right = RightRotate(pRoot->p_right);
            pRoot = LeftRotate(pRoot);
        }
    }
```

❖ Hàm xóa một node: ***void Remove(NODE* &pRoot, int x)***

- Thực hiện thao tác xóa như cây nhị phân, sau đó thực hiện cập nhật chiều cao cây và cân bằng cây nếu cây mất cân bằng tại các trường hợp: L-L, L-R, R-L, R-R;

```
if (pRoot == nullptr) return;
    updateHeight(pRoot);
```

```
int check = getHeight(pRoot->p_left) - getHeight(pRoot->p_right);
if(check > 1){
        int check_2 = getHeight(pRoot->p_left->p_left) -
getHeight(pRoot->p_left->p_right);
        if(check_2 >= 0) pRoot = RightRotate(pRoot);
        else {
                pRoot->p_left = LeftRotate(pRoot->p_left);
                pRoot = RightRotate(pRoot);
        }
}
if(check < -1){
        int check_2 = getHeight(pRoot->p_right->p_left) -
getHeight(pRoot->p_right->p_right);
        if(check_2 <= 0) pRoot = LeftRotate(pRoot);
        else {
                pRoot->p_right = RightRotate(pRoot->p_right);
                pRoot = LeftRotate(pRoot);
        }
}
```

❖ Hàm kiểm tra cây AVL: ***bool isAVL(NODE* pRoot);***

- Kiểm tra độ chênh lệch chiều cao các nhánh trái phải của cây, để xác định, rồi đệ quy để kiểm tra tại từng cành.

```
if(pRoot == nullptr)return true;
    int check = getHeight(pRoot->p_left) - getHeight(pRoot->p_right);
    if(check > 1 || check <-1) return false;
    return isAVL(pRoot->p_left) && isAVL(pRoot->p_right);
```