REPORT_24120184

*Binary_tree:

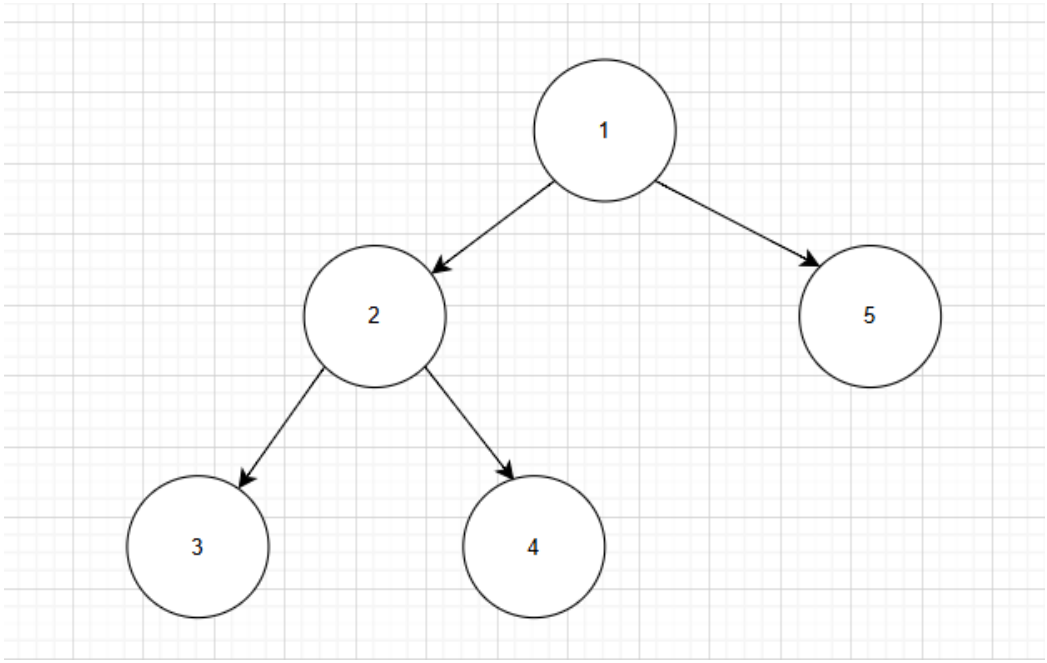
1.Hàm NODE* createNode(int data):

- Tạo 1 node mới có giá trị là data, dựa vào cấu trúc của node:
- Cho key của node mới bằng data , left và right của node mới sẽ trỏ đến null.
- Trả về giá trị node mới.

2. Hàm vector<int> NLR(NODE* pRoot):

- Ta duyệt cây theo thứ tự node -> left -> right , bằng cách sử dụng đệ quy
- Đầu tiên ta sẽ thêm giá trị của node vào trong vector do ở vị trí node, gọi đệ quy đối với node bên trái sau đó là node bên phải.

VD:



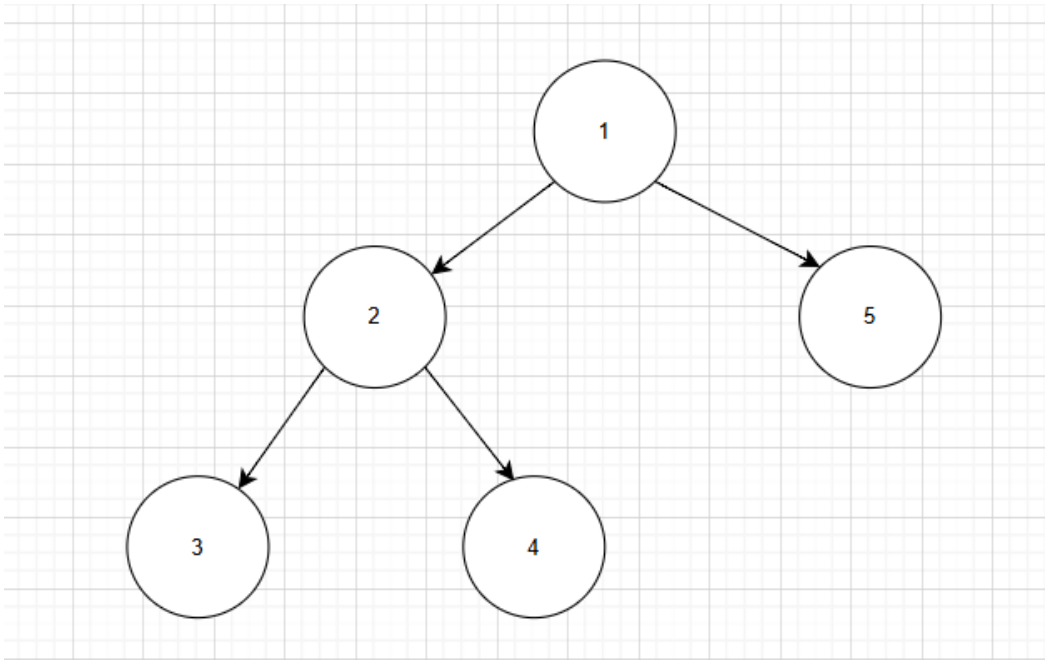
- Ở đây vì lưu giá trị trước bước đệ quy trái sau khi đệ quy bên trái đến node cuối bên trái tức là node (3) thì vector đã lưu được (1 , 2 , 3).
- Thoát các hàm đệ quy trái và ta đệ quy phải theo thứ tự => Ở bước đệ quy node (2) ta đệ quy trái đến đệ quy phải chính là node (4) , ở bước đệ quy node (1) sau khi đệ quy node (2) ta đệ quy node (5)

⇨ Kết quả thu được: (1,2,3,4,5).

3.Hàm vector<int> LNR(NODE* pRoot):

- Ta duyệt cây theo thứ tự left => node => right, bằng cách sử dụng đệ quy.
- Vì duyệt bên trái trước nên ta đệ quy node bên trái sau đó mới tiến hành lưu giá trị ,sau đó tiến hành đệ quy node bên phải.

VD:



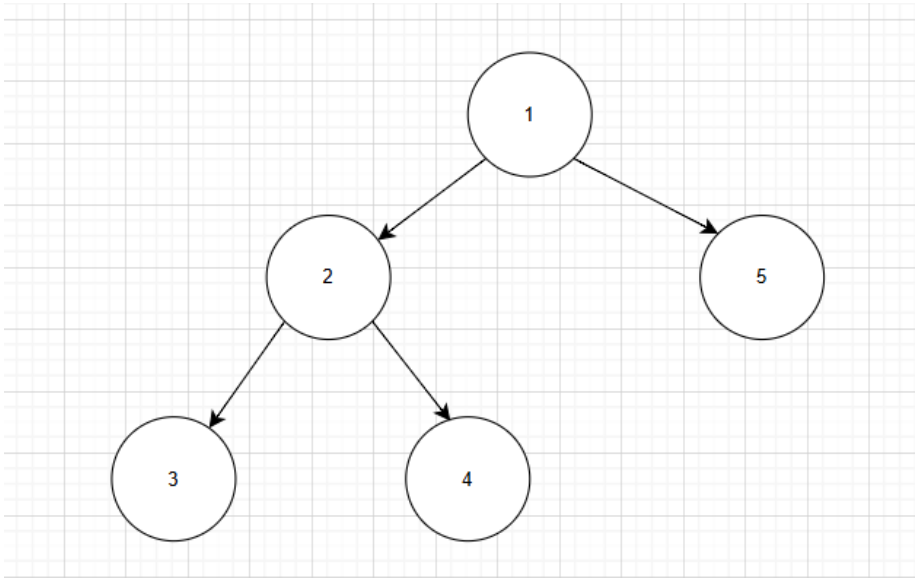
- Ở đây ta đệ quy node sau cùng bên trái tức là đệ quy từ node(1) => node(2) => node(3) vì đã là node con bên trái cuối nên tiến hành lưu (3) sau đó quay về bước đệ quy ở node (2) lưu (2) tiến hành đệ quy bên trái phải (4) , quay về bước đệ quy ở node (1) lưu (1) sau cùng là lưu node bên trái node (5)

⇨ Kết quả thu được: (3, 2, 4, 1, 5).

4.Hàm vector<int> LRN(NODE* pRoot):

- Ta duyệt cây theo thứ tự left => right => rnode, bằng cách sử dụng đệ quy.
- Ở đây ta cần đệ quy cả bên trái và bên phải sau đó mới lưu giá trị.

VD:



- Đầu tiên ta đệ quy trái đến được node (3) và lưu giá trị node (3), sau đó quay ngược về node (2) tiến hành đệ quy phải tức là node (4) lưu giá trị node (4) ,sau đó quay

ngược về node (2) lưu giá trị node (2) tiếp tục quay về node (1) tiến hành đệ quy bên phải chính là node (5) (lưu giá trị) quay ngược về node (1) và lưu.

⇨ Kết quả thu được: (3,4,2,5,1).

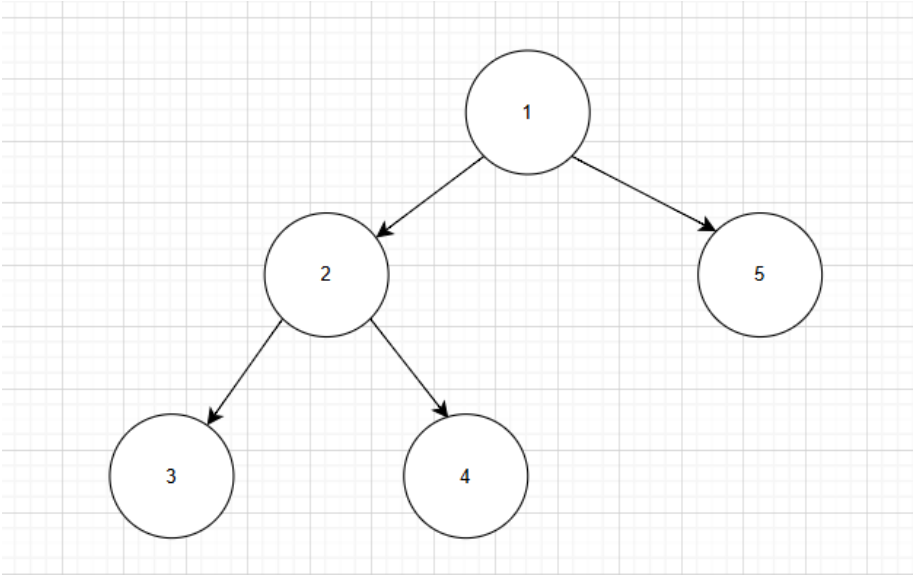
5.Hàm vector<vector<int>> LevelOrder(NODE* pRoot):

- Duyệt cây theo chiều rộng ta duyệt theo từng tầng của cây theo thứ tự từ trái qua phải.(BFS)
- Ý tưởng sử dụng queue để thực hiện:
- Ta lần lượt bỏ các node ở từng tầng vào queue:

  -Ở ban đầu vì chỉ có 1 node nên ta thêm vào queue, tiến hành lưu giá trị trả về kết quả duyệt, nếu node trái hoặc node phải của queue không rỗng ta tiến hành thêm vào queue để cho lượt duyệt ở tầng tiếp theo

  -Sau khi lưu và thêm vào queue ta bỏ node ta đang xét ra khỏi queue làm lần lượt cho đến khi queue rỗng.

VD:



- Ở tầng 0: Thêm node 1 vào queue: Tiến hành lưu -> Thêm 2 vào queue, thêm 5 vào queue -> Xóa 1 khỏi queue.
- Ở tầng thứ 1: Queue hiện tại (2 , 5):

  -Xét 2: Lưu giá trị 2 -> Thêm 3 và 4 vào queue -L> Xóa 2 khỏi queue.

  -Xét 5: Lưu giá trị 5 -> Không thêm do không có node trái và node phải -> Xóa 5 khỏi queue.

- Ở tầng 2: Queue hiện tại (3 , 4):

  -Vì cả 3 và 4 đều không có node con trái và node con phải nên ta lưu lần lượt 3 và 4 đồng thời xóa khỏi queue.

- Queue rỗng => Dừng lại

=> Kết quả thu được : Tầng 0 : 1 , Tầng 1: 2, 5 Tầng 2: 3,4.

6.Hàm countNode(NODE* pRoot):

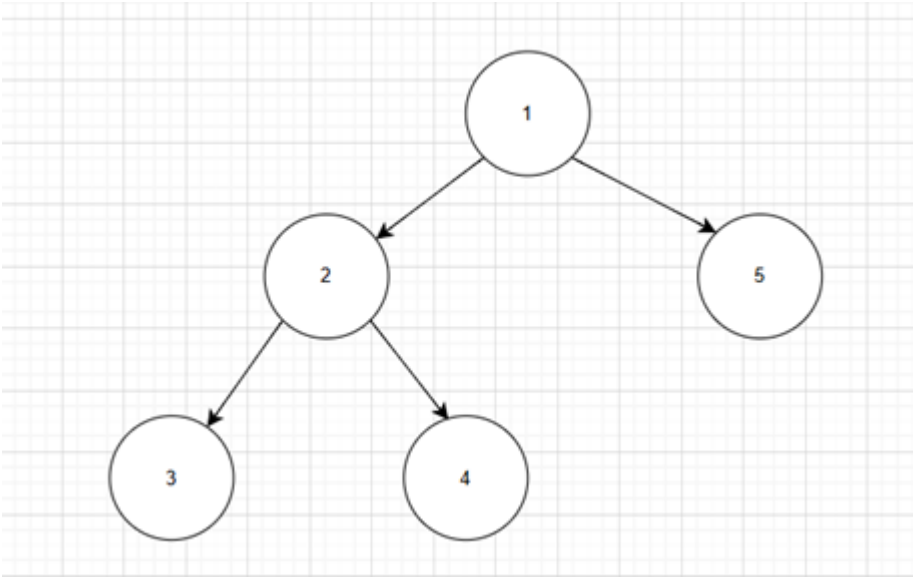
- Ta cần đếm số lượng Node có trong cây, ta sẽ làm tương tự như duyệt theo chiều rộng bằng cách sử dụng queue.
- Nhưng sẽ khác ở chỗ thay vì ta lưu giá trị của node thì biến đếm sẽ tăng thêm 1 đơn vị.

7.Hàm int sumNode(NODE* pRoot):

- Ta cần tính tổng của các Node có trong cây, tương tự hàm countNode mỗi lần duyệt ta chỉ cần cộng giá trị của node đó vào biến tổng.

8.Hàm heightNode(NODE* pRoot, int value):

- Tìm chiều cao của Node có giá trị là value, nếu không có ta trả về -1, ta sử dụng đệ quy tìm chiều cao của node hiện tại dựa vào chiều cao của node bên trái và node bên phải.
- Chiều cao của node bằng max(chiều cao node trái, chiều cao node phải) + 1.
- Cụ thể:



- Để tìm chiều cao node (1), ta cần biết chiều cao node (2) và node (5)
- Chiều cao node (2) bằng chiều cao của node (3) hoặc node (4) (do node (3) và node (4) có cùng chiều cao) + 1 = 2.
- Chiều cao của node (5) bằng 1 (do không có node con)

⇨ Chiều cao của node (1) = chiều cao node (2) (chiều cao node (2) lớn hơn node (5)) + 1 = 3

9.Hàm Level(NODE* pRoot, NODE* p):

- Hàm giúp tìm Level của 1 node cụ thể , ta tìm level thông qua đệ quy (đệ quy lần lượt bên trái và bên phải để đi qua tất cả các node) cứ mỗi lần đi xuống 1 node thì Level ở hiện tại sẽ cộng thêm cho 1, cho đến khi ta tìm được node cần tìm thì gán kết quả của Level cần trả về với Level hiện tại đang xét.

10.Hàm countLeaf(NODE* pRoot):

- Hàm giúp đếm số lượng lá có trong 1 cây. Ta sử dụng đệ quy cho node bên trái và bên phải đệ duyệt các node
- Nếu node đang xét hiện tại không có node con tức là node bên trái và phải trống thì node đó chính là node lá, biến đếm tăng thêm 1.
- Kết quả cuối cùng trả về số lá có trong cây.

*bst (Binary_search_tree):

1.Hàm NODE* Search(NODE* pRoot, int x):

- Tìm node có giá trị x là trong 1 cây thông qua hàm search, cây nhị phân tìm kiếm có tính chất trong mỗi node thì tất cả node con bên trái sẽ có giá trị nhỏ hơn giá trị của node, và tất cả node con bên phải có giá trị lớn hơn giá trị của node.
- Ta có thể gán điều kiện để có thể tìm node có giá trị là x một cách nhanh hơn nếu nó đã đảm bảo là 1 cây nhị phân tìm kết, nhưng ở đây chưa rõ nên chỉ xét trường hợp tổng quát:
- Bằng cách gọi đệ quy cây con bên trái và bên phải ta sẽ duyệt được tất cả các node của cây, cho đến khi node đang xét có giá trị bằng x thì ta trả về node đó.

2.Hàm Insert(NODE* &pRoot, int x):

- Thêm 1 Node có giá trị là x vào cây nhị phân tìm kiếm, ta sử dụng đệ quy để chèn node:

  -Nếu x lớn hơn giá trị của node đang xét thì ta tiến hành đệ quy bên trái.

  -Nếu x nhỏ hơn giá trị của node đang xét thì ta tiến hành đệ quy bên phải.

  -Nếu x bằng giá trị node đang xét thì ta thoát khỏi hàm do không thể chèn 1 node có cùng giá trị với node đã có sẵn trong cây.

  -Cho đến khi tìm được vì trí phù hợp thì ta sẽ chèn node đó vào.

3.Hàm void Remove(NODE* &pRoot,int x):

- Hàm xóa 1 node có giá trị là x trong 1 cây, dựa vào tính chất của cây nhị phân tìm kiếm ta sử dụng đệ quy với các điều kiện sau:

  -Nếu x lớn hơn giá trị của node đang xét thì ta tiến hành đệ quy bên trái.

  -Nếu x nhỏ hơn giá trị của node đang xét thì ta tiến hành đệ quy bên phải.

  -Nếu x bằng giá trị node đang xét thì node đang xét chính là node cần xóa:

  *Đến đây ta lại chia làm các trường hợp:

  +node cần xóa không có node con thì ta chỉ cần xóa và gán nó bằng nullptr.

  +node cần xóa chỉ có 1 node con thì ta xóa node cần xóa, cho node con đó thế chỗ node vừa mới xóa.

+node cần xóa có 2 con thì ta có thể thay node đó bằng nó lớn nhất bên trái hoặc node nhỏ nhất bên phải. (ở đây ta chỉ cần cho node hiện tại bằng giá trị của node con cần chèn sau đó xóa node chèn đó bằng đệ quy).

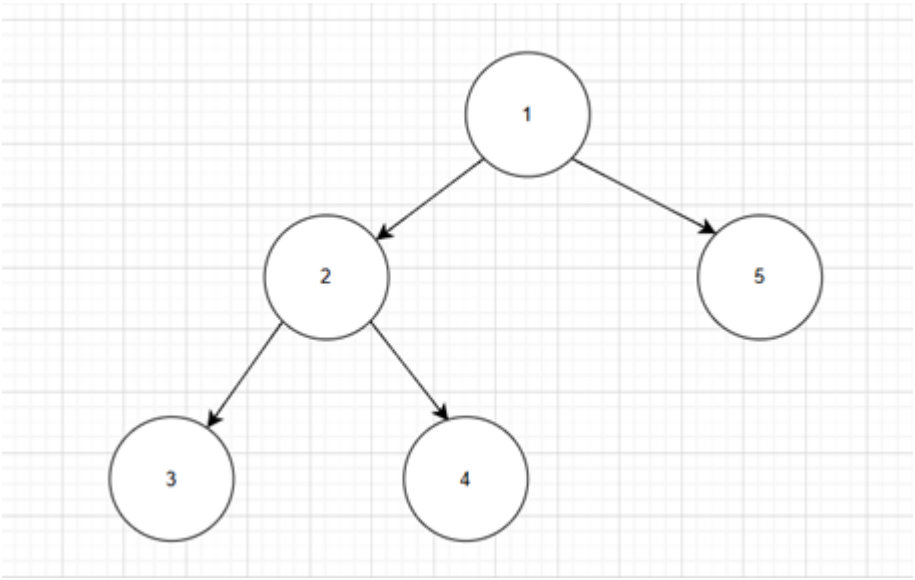
4.Hàm createTree(int a[], int n):

- Hàm này tạo ra cây đệ quy với các giá trị dựa vào mảng a, ta duyệt qua các phần tử của mảng a và thêm nó vào trong cây bằng hàm Insert đã viết.

5. Hàm void removeTree(Node* &pRoot):

- Mục đích của hàm nhằm xóa hết tất cả các node trong cây, ý tưởng dùng đệ quy truy cập đến node con chính là các node lá xóa lần lượt cho đến ngọn và đến khi không còn node để xóa nữa.

VD:



- Ta sẽ xóa các node lá:
- Sau khi xóa 3, 4, 5 thì node 2 lúc này cũng trở thành node lá => Sau khi xóa 2 thì xóa node 1 cũng chính là node lá.

6. int Height(NODE* pRoot):

- Hàm dùng để tính chiều cao của 1 cây, tương tự ở hàm heightNode ta tính được chiều cao của 1 node dựa vào chiều cao của 2 node con trái và phải thông qua đệ quy, để tính chiều cao của cây thì node ta cần tính chiều cao chính là node gốc.

7. int countLess(NODE* pRoot, int x):

- Đếm số lượng node có giá trị nhỏ hơn x bằng cách gọi đệ quy:

  -Nếu node rỗng thì ta trả về 0.

  -Nếu node hiện tại nhỏ hơn x:

+Đếm node này (+1)

+Và đệ quy kiểm tra cả trái và phải

-Nếu node hiện tại lớn hơn hoặc bằng x:

+Không đếm node này

+Chỉ kiểm tra cây con bên trái (vì các node bên phải đều lớn hơn hoặc bằng node hiện tại).

8.Hàm int countGreater(NODE* pRoot, int x):

- Đếm số lượng node có giá trị lớn hơn x bằng cách gọi đệ quy:

-Nếu node rỗng → trả về 0.

-Nếu node hiện tại lớn hơn x:

+Đếm node này (+1)

+Và đệ quy kiểm tra cả trái và phải

-Nếu node hiện tại nhỏ hơn hoặc bằng x:

+Không đếm node này

+Chỉ kiểm tra cây con bên trái (vì các node bên phải đều nhỏ hơn hoặc bằng node hiện tại).

9.Hàm bool isBST(NODE* pRoot):

- Hàm giúp ta kiểm tra 1 cây có phải là cây tìm kiếm hay không, ta đệ quy duyệt cây con bên trái và cây con bên phải. Điều kiện để mỗi node thỏa là:

-Giá trị node hiện tại phải lớn hơn giá trị lớn nhất của cây con bên trái

-Và nhỏ hơn giá trị nhỏ nhất của cây con bên phải

-Nếu tất cả các node đều thỏa mãn điều kiện trên, thì đó là một cây BST hợp lệ.

- Có 2 hàm bổ sung tìm max cho cây con bên trái và tìm min cho cây con bên phải.

10.Hàm bool isFullBST(NODE* pRoot):

- Kiểm tra xem một cây nhị phân có phải là cây nhị phân đầy đủ (full binary tree) hay không.
- Cây nhị phân đầy đủ là cây mà mỗi cây con không con nào hoặc có đúng 2 node con.
- Xét các trường hợp:

-Trường hợp node rỗng tức là không có cây con ta trả về true.

-Kiểm tra node nếu chỉ có cây con trái hoặc cây con phải thì trả về false.

-Đệ quy kiểm tra các node còn lại , đệ quy bên trái và bên phải.

-Nếu đã đủ phần tử và thỏa hàm isBST thì đó là cây nhị phân tìm kiếm đầy đủ.

*AVL Tree:

1.Hàm NODE* createNode(int data):

- Tương tự hàm createNode ở trên nhưng có thên chiều cao height và cho chiều cao height bằng 0.

Hàm insert và hàm remove ta viết hàm cũng gần giống như hàm insert và hàm remove của bst nhưng ta cần kiểm tra cây cần kiểm tra từng node có thỏa điều kiện hay không và sau đó cân bằng. Nên có 1 số hàm phụ trợ:

int Height(NODE* node):

- Trả về chiều cao của node đang xét nếu là null thì trả về 0.

int GetBalance(NODE* node):

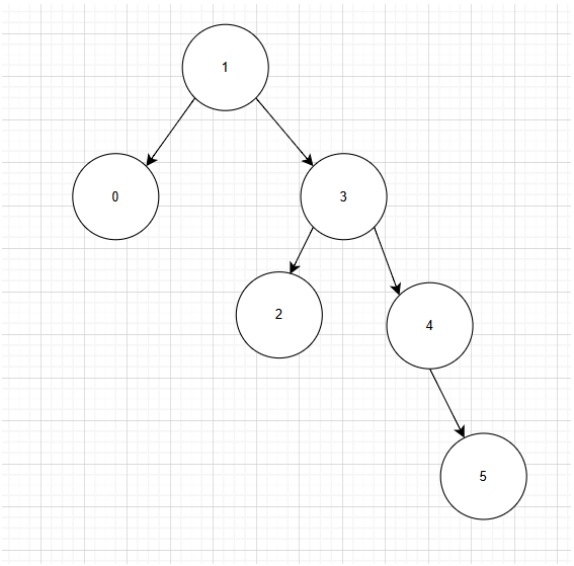
- Giá trị cân bằng của node (trạng thái cân bằng là chiều cao của node bên trái và bên phải chênh lệch không quá 1).

void updateHeight(NODE* node)

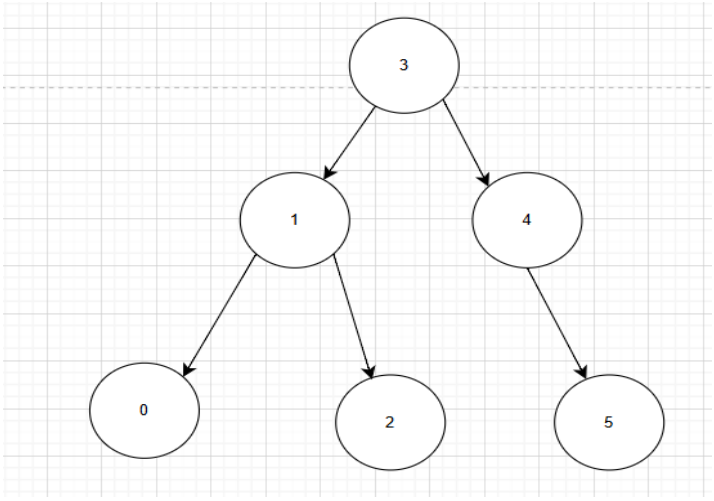
- Sau khi thực hiện các thao tác quay, chèn , xóa ta cần cập nhật lại chiều cao của node sao cho đúng.
- Chiều cao của node được tính bằng chiều cao lớn hơn giữa node trái và phải cộng thêm cho 1.

void LeftRotate(NODE* &node):

- Xoay trái có tác dụng điều chỉnh trạng thái cân bằng của node.
- Ở đây node là node không cân bằng.
- Gọi node bên phải của node là node1, node bên trái của node1 là Lnode1.
- Bước xoay: cho left của node1 là node, cho right của node là Lnode1.
- Cập nhật chiều cao của node và node1, trả về đúng vị trí node gốc.
- Xét ví dụ:



- Ở đây node (1) đang mất cân bằng do node bên trái có chiều cao là 0, node bên phải có chiều cao là 2 => dẫn đến mất cân bằng ta tiến hành xoay trái:
- node1 ở đây là node (3) , Lnode1 là node (2)
- Ta cho left của node1 là node và cho right của node là Lnode1 ta thu được:



⇨ Lúc này đã cân bằng , ta cập nhật chiều cao và trả về node gốc là node (3).

Tương tự với xoay trái ta có xoay phải: void RightRotate(NODE* &node):

- Gọi node bên trái của node là node1, node bên phải của node1 là Rnode1.
- Bước xoay: cho right của node1 là node, cho left của node là Rnode1.
- Cập nhật chiều cao của node và node1, trả về đúng vị trí node gốc.

Hàm balance(NODE* &node):

- Hàm giúp cân bằng node đang không cân bằng:
- Đầu tiên ta cần updateHeight: Cập nhật lại chiều cao của node hiện tại. Việc này rất cần thiết vì sau mỗi thao tác chèn hoặc xóa node, chiều cao có thể thay đổi.
- Qua hàm GetBalance để kiểm tra node có mất cân bằng hay không:

  -Tính hệ số cân bằng của node = chiều cao bên trái - chiều cao bên phải.

  -Nếu > 1 thì lệch trái.

  -Nếu < -1 thì lệch phải.

  -Nếu nằm trong khoảng từ -1 đến 1 thì node đã cân bằng, không cần làm gì.

- Xét các trường hợp mất cân bằng:

  -Trường hợp 1: Lệch trái (> 1):

  +Ta kiểm tra GetBalance(node->p_left) nếu nhỏ hơn 0 thì con trái lệch phải: cần xoay kép => xoay trái trước sau đó xoay phải.

  +Nếu con trái không lệch phải thì ta chỉ cần xoay phải.

  -Trường hợp 2: Lệch phải (< 1):

+Ta kiểm tra GetBalance(node->p_right) nếu lớn hơn 0 thì con phải lệch trái: xoay kép => xoay bên phải trước sau đó xoay trái.

+Nếu không lệch trái ta chỉ cần xoay trái.

2.Hàm void Insert(NODE* pRoot,int x)

- Ta làm tương tự như hàm insert của bst nhưng thêm vào hàm balance nhằm đảm bảo tất cả node đều cân bằng.

3.Hàm void Remove(NODE* &pRoot,int x):

- Ta làm tương tự như hàm Remove của bst nhưng thêm vào hàm balance nhằm đảm bảo tất cả node đều cân bằng.

4.Hàm bool isAVL(NODE* pRoot)

- Ta cần kiểm tra tất cả các node có cân bằng hay không bằng cách đệ quy trái và phải dể đi qua tất cả các node, đồng thời nếu là cây cân bằng thì đó cũng cần là cây nhị phân tìm kiếm (thêm hàm isBST).

Hình ảnh minh chứng up lên Github: