**Báo cáo Project – Tuần 7**

**I. Binary Tree:**

1. NODE\* createNode(int data): Tạo 1 Node mới.

Để tạo Node mới, cho key của Node này có giá trị là Data, 2 nhánh left, right của Node trỏ vào nullptr.

2. vector<int> NLR(NODE\* pRoot): Duyệt cây theo thứ tự NLR và đưa các phần tử này vào vector.

Duyệt NLR tức là duyệt theo thứ tự Node-Left-Right. Ở đây cần dùng đệ quy với base case là trả về vector khi Node rỗng.

- Đầu tiên, ta duyệt qua Node và đưa phần tử này vào vector.

- Tiếp theo, đệ quy duyệt các phần tử bên trái của Node vừa xét và đẩy các phần tử này vào vector.

- Sau đó, đệ quy duyệt các phần tử bên phải của Node vừa xét và đẩy các phần tử này vào vector.

3. vector<int> LNR(NODE\* pRoot): Duyệt cây theo thứ tự LNR và đưa các phần tử này vào vector.

Duyệt LNR: Duyệt theo thứ tự Left-Node-Right.

Base case: Trả về vector khi Node rỗng. Các bước duyệt:

- Dùng đệ quy để duyệt và đẩy các phần tử bên trái Node vào vector.

- Duyệt và đẩy Node vào vector.

- Dùng đệ quy duyệt và đẩy các phần tử bên phải Node vào vector.

4. vector<int> LRN(NODE\* pRoot): Duyệt cây theo thứ tự LRN và đưa các phần tử này vào vector.

Duyệt LRN: Duyệt theo thứ tự Left-Right-Node.

Base case: Trả về vector khi Node rỗng. Các bước duyệt:

- Dùng đệ quy duyệt và đẩy các phần tử bên trái Node vào vector.

- Dùng đệ quy duyệt và đẩy các phần tử bên phải Node vào vector.

- Duyệt và đẩy Node vào vector.

5. vector<vector<int>> LevelOrder(NODE\* pRoot): Duyệt theo thứ tự BFS – từ trái qua phải của cây theo từng level.

Vì trong hàm này cần dùng đệ quy để duyệt nên ta cần dùng 1 hàm phụ để thuận tiện cho việc đệ quy. Hàm phụ này có chức năng lưu các phần tử của cây vào vector theo level hợp lý. Hàm này sẽ gồm cây cần duyệt, 1 biến level để lưu các cấp và 1 vector các vector (1 vector lớn gồm nhiều vector con). Ứng với mỗi số thứ tự của vector con trong vector lớn sẽ là level của Node.

Base case: Nếu cây rỗng -> Rời khỏi hàm.

- Với mỗi lần sắp duyệt, kiểm tra xem cấp của cây có vượt số vector trong vector chưa. Nếu đã vượt, phải bổ sung thêm 1 vector trong vector lớn nữa.

- Duyệt theo kiểu NLR: Thêm phần tử Node đang duyệt vào vector con đang xét.

Đệ quy qua Left của Node đang xét, mỗi lần qua thành công sẽ tăng level lên 1. Sau khi đã đệ quy xong Left, đệ quy qua Right của Node đang xét, mỗi lần đệ quy thành công sẽ tăng level lên 1. (vì Left, Right của Node đang xét có level cao hơn Node đó là 1).

- Sau khi thiết lập xong hàm phụ, trong hàm chính ta tạo 1 vector của vector, rồi cho vector này xử lý qua hàm phụ với level bắt đầu từ 0. Sau đó, trả về vector này.

6. int countNode(NODE\* pRoot): Đếm số lượng Node trong cây.

Số lượng Node của cây cũng chính là số lương phần tử trong vector khi duyệt qua cây. Ta duyệt vector NLR (hoặc LNR, LRN) cho cây rồi trả về số phần tử của vector.

7. int sumNode(NODE\* pRoot): Tính tổng các Node trong cây.

Tạo 1 biến sum = 0 là tổng của các Node trong cây. Duyệt vector cho cây theo kiểu NLR (hoặc LNR, LRN). Duyệt qua tất cả các phần tử của vector. Với mỗi lần duyệt, cộng phần tử đang duyệt vào sum. Sau đó, trả về sum.

8. int heightNode(NODE\* pRoot, int value): Tính độ cao của 1 Node chứa giá trị cho trước.

- Base case:

+ Nếu không có giá trị cho trước trong cây -> trả về -1.

+ Nếu có giá trị cần tìm: Tạo 2 biến chiều cao bên trái và bên phải.

Kiểm tra xem nhánh bên trái Node có nullptr chưa. Nếu chưa, tiếp tục đệ quy tính chiều cao Node ở bên trái. Tương tự với nhánh bên phải. Tiếp tục làm như vậy cho tới khi chạm tới Node nullptr thì trả về -1 rồi quay lui lại.

Sau khi duyệt xong, kiếm tra chiều cao của các Node bên trái và bên phải. Chiều cao của Node cha sẽ lấy chiều cao lớn nhất của Node con trái hoặc Node con phải + 1. Do đó, để tính chiều cao của Node cha, ta chỉ cần lấy chiều cao lớn nhất của một trong hai Node con cộng thêm 1. Để tính chiều cao của các Node con thì ta đã thực hiện trong việc đệ quy (như trên).

+ Nếu chưa tìm được giá trị: Tiếp tục đệ quy tới các Node bên trái và phải.

9. int Level(NODE\* pRoot, NODE\* p): Tính cấp của Node đang xét.

Trong hàm này, cần dùng lại hàm LevelOrder đã xét ở trên. Ta duyệt qua từng phần tử của từng vector nhỏ của vector lớn. Duyệt cho tới khi tìm được phần tử -> trả về số thứ tự của hàng vector đang xét, tương ứng với level của Node.

10. int countLeaf(NODE\* pRoot): Đếm số lá của cây.

- Base case:

+ Nếu cây rỗng -> Cây không có lá.

+ Nếu cây chỉ có 1 phần tử -> Cây có 1 lá.

- Recursion case:

Cây có 2 phần tử trở lên: Dùng đệ quy để đếm số lá của cây con bên trái, và số lá của cây con bên phải, sau đó cộng lại.

**II. Binary Search Tree:**

1. NODE\* Search(NODE\* pRoot, int x): Tìm kiếm 1 phần tử trong cây.

- Base case: Nếu cây rỗng -> Trả về nullptr.

- Recursion case:

+ Nếu x == giá trị của Node đang xét -> Trả về Node.

+ Nếu x > giá trị của Node đang xét -> Node cần tìm nằm bên phải -> Đệ quy duyệt qua bên phải.

+ Nếu x < giá trị của Node đang xét -> Node cần tìm nằm bên trái -> Đệ quy duyệt qua bên trái.

2. void Insert(NODE\* &pRoot, int x): Thêm Node vào cây.

- Base case: Nếu cây rỗng -> Thêm phần tử vào Root của cây.

- Recursion case:

+ Nếu x > Node đang xét -> x cần thêm vào bên phải cây -> Đệ quy thêm qua bên phải.

+ Nếu x < Node đang xét -> x cần thêm vào bên trái cây -> Đệ quy thêm qua bên trái.

+ Nếu x == Node đang xét -> Không cần thêm.

3. void Remove(NODE\* &pRoot, int x): Xoá 1 Node trong cây.

\* Base case: Nếu cây rỗng -> Không cần xoá, trả về cây cũ.

\* Recursion case:

- Nếu x > Node đang xét -> Giá trị cần xoá ở bên phải cây -> Đệ quy duyệt qua bên phải.

- Nếu x < Node đang xét -> Giá trị cần xoá bên trái cây -> Đệ quy duyệt qua bên trái.

- Nếu x == Node đang xét -> Xoá Node này:

+ Nếu Node này là Node lá -> Xoá ngay.

+ Nếu Node này có 1 nốt con trái (hoặc phải) -> Cho Node ở trước kề nó liên kết với Node con trái (hoặc phải) kề sau nó, sau đó xoá Node này.

+ Nếu Node này có 2 nốt con: Xoá Node ở bên phải cùng của cây con trái (hoặc bên trái cùng của cây con phải). Trong code, em xoá Node bên phải cùng của cây con trái. Cách xoá: Cho 1 Node chạy đi tìm Node bên phải cùng của cây con trái (gọi là temp) và 1 Node khác đi liên trước Node cần xoá (holder). Sau đó, đổi giá trị của Node cần xoá với Node gốc đang xét. Nếu như holder không chạy được -> Node cần xoá là Node kề trái của Node đang xét -> Xoá đi như cách xoá 1 Node có 1 nốt con trái. Nếu như holder chạy được -> Cho bên phải holder trỏ vào bên trái của temp (Nếu như temp->left == nullptr thì temp chính là Node lá -> xoá tương tự như trường hợp Node lá. Nếu temp->left != nullptr thì temp không là Node lá -> Xoá tương tự trường hợp 1 Node lá).

4. NODE\* createTree(int a[], int n): Tạo cây từ mảng đã cho.

Duyệt các phần tử trong mảng. Qua mỗi lần duyệt, thực hiện chèn phần tử vào cây.

5. void removeTree(Node\* &pRoot): Xoá toàn bộ cây.

- Base case: Nếu cây rỗng -> Trả về cây cũ.

- Recursion case: Nếu cây không rỗng: Lần lượt đệ quy xoá cây con bên trái, xoá cây con bên phải rồi xoá Node gốc. Sau đó, cho cây trả về nullptr để đảm bảo đã được xoá.

6. int Height(NODE\* pRoot): Đo chiều cao của cây.

- Base case: Nếu cây rỗng -> Trả về -1.

- Recursion case:

+ Tạo 1 biến đếm, đệ quy đo chiều cao bên trái, 1 biến đếm khác đệ quy đo chiều cao bên phải (so với Node đang xét).

+ Nếu chiều cao bên trái lớn hơn bên phải -> Độ cao của cây = Chiều cao bên trái + 1. Nếu ngược lại, độ cao của cây = Chiều cao bên phải + 1.

7. int countLess(NODE\* pRoot, int x): Đếm số lượng Node trong cây có giá trị thấp hơn x.

- Base case: Nếu cây rỗng -> Không có phần tử -> Trả về 0.

- Recursion case:

+ Nếu x <= Node đang xét -> Các Node bên phải luôn lớn hơn x, vì thế không xét nữa -> Đệ quy đếm qua bên trái của cây để đếm các Node nhỏ hơn x.

+ Nếu x > Node đang xét -> Các Node bên trái luôn nhỏ hơn x -> Đệ quy đếm qua bên phải của cây để đếm các Node nhỏ hơn x, rồi cộng các Node bên trái (vì các Node này đã luôn nhỏ hơn x), cộng thêm 1 (Node đang xét nhỏ hơn x).

8. int countGreater(NODE\* pRoot, int x): Đếm số lượng Node trong cây có giá trị lớn hơn x.

- Base case: Nếu cây rỗng -> Không có phần tử -> Trả về 0.

- Recursion case:

+ Nếu x >= Node đang xét -> Các Node bên trái luôn nhỏ hơn x -> Đệ quy đếm qua bên phải cây.

+ Nếu x < Node đang xét -> Các Node bên phải luôn lớn hơn x -> Số Node = Số lượng Node bên phải + Số lượng Node bên trái mà lớn hơn x (sẽ đếm được bằng đệ quy) + 1(Node đang xét lớn hơn x).

9. bool isBST(NODE\* pRoot): Kiểm tra xem 1 cây có phải là cây BST không.

Để kiểm tra, ta duyệt LNR cây rồi lưu qua 1 vector. Nếu các phần tử này tăng dần thì cây này là cây BST.

10. bool isFullBST(NODE\* pRoot): Kiểm tra xem 1 cây có phải là cây BST đầy đủ không.

Cây BST đầy đủ là cây mà mọi Node có 0 hoặc 2 nốt con.

Để là cây BST đầy đủ, cây này cần phải là cây BST và có mọi Node 0 hoặc 2 nốt con (cây đầy đủ). Cách kiểm tra cây BST như trên.

Để kiểm tra xem cây có phải là cây đầy đủ không, ta làm như sau:

- Kiểm tra nếu cây rỗng -> là cây đầy đủ.

- Nếu Node đang xét của cây chỉ có 1 nốt con trái hoặc phải -> không là cây đầy đủ, false.

- Nếu Node đang xét có cả 2 nốt con -> Đệ quy kiểm tra tiếp các nốt con trái và phải.

**III. AVL Tree:**

1. NODE\* createNode(int data): Tạo 1 Node mới.

Mỗi Node mới sẽ có giá trị là data, left và right trỏ tới nullptr, ngoài ra còn có chiều cao khởi đầu là 1.

Trước khi đi đến các hàm tiếp theo, giải thích một chút về các hàm phụ:

- Hàm height: Tính chiều cao của Node: là 0 nếu Node rỗng, 1 nếu Node không rỗng.

- Hàm updateHeight: Cập nhật chiều cao của Node. Node mới sẽ có chiều cao là Node kề sau nó cộng 1. Nếu có 2 Node con, lấy Node có chiều cao lớn hơn rồi cộng 1.

- Hàm diffHeight: Tính chênh lệch chiều cao của bên trái và phải.

- Hàm leftRotation, rightRotation: Quay trái, quay phải Node trong cây.

2. void Insert(NODE\* &pRoot, int x): Thêm 1 phần tử vào cây.

\* Nếu như cây rỗng -> Thêm Node vào Root của cây.

\* Nếu cây không rỗng, cách thêm giống như thêm phần tử trong cây BST. Tuy nhiên, sau khi thêm, có thể cây bị mất cân bằng -> Cần:

- Câp nhật lại vị trí chiều cao của cây.

- Tính chênh lệch giữa trái và phải. Nếu <= 1 thì đã là cây AVL. Nếu > 1 thì cần xét:

+ Nếu lệch về phía trái và x < left của Node đang xét -> Lệch trái trái -> Cần xoay phải.

+ Nếu lệch về phía phải và x > right của Node đang xét -> Lệch phải phải -> Cần xoay trái.

+ Nếu lệch về phía trái và x > left của Node đang xét -> Lệch trái phải -> Cần xoay trái của left ở Node đang xét, rồi xoay phải ở Node đang xét.

+ Nếu lệch về phía phải và x < right của Node đang xét -> Lệch phải trái -> Cần xoay phải của right ở Node đang xét, rồi xoay trái ở Node đang xét.

Sau khi xoay xong -> Cây AVL.

3. void Remove(NODE\* &pRoot, int x): Xoá 1 Node trong cây AVL.

Cách xoá tương tự như cây BST. Tuy nhiên, sau khi xoá, cây có thể bị mất cân bằng -> Cần cập nhật lại để đúng dạng cây AVL.

Cách xoay cũng tương tự như hàm Insert. Tuy nhiên, thay vì so sánh phần tử với Node, ta sẽ so sánh chiều cao của Node trái với Node phải.

4. bool isAVL(NODE\* pRoot): Kiểm tra xem cây có phải là cây AVL không.

Để là cây AVL, cây phải là 1 cây BST và có chiều cao bên trái và phải không chênh nhau quá 1.

Cách kiểm tra cây BST đã làm ở trên. Ta cần kiểm tra xem chiều cao bên trái và phải của cây có chênh nhau quá 1 không.

Ta tính độ chênh tại Node đang xét. Nếu chênh nhau quá 1 -> không phải cây AVL. Nếu chênh nhau không quá 1 -> Đệ quy xét tiếp các Node bên trái và phải của cây.