BÀI BÁO CÁO THUẬT TOÁN DSA – WEEK 7

Nguyễn Đình Trung Kiên – 24120195

I. BINARY TREE:

1. Hàm phụ:

1.1. vectorConcat:

- Hàm nối 2 vector lại với nhau (vector thứ hai vào đuôi vector thứ nhất).

1.2. findNode:

- Sử dụng đệ quy đến 2 nhánh của từng NODE để tìm NODE có giá trị *val.*

1.3. treeHeight:

- Sử dụng đệ quy để tính độ cao của từng NODE, ta sẽ gọi đệ quy để tính độ cao của 2 cây con trái và phải của NODE đang xét và sẽ lấy max của hai độ cao đó.

2. Hàm chính:

2.1. createNode:

- Trả về con trỏ đến một NODE mới mang giá trị *data* được cung cấp, gán NULL cho 2 con trỏ *p_left* và *p_right* để tạo thành một NODE hợp lệ.

2.2. NLR:

- Sử dụng đệ quy để duyệt Tree theo kiểu NLR (Pre-order).
- Ta lần lượt gọi hàm đệ quy với cây con bên trái của *pRoot* để nối giá trị ở *pRoot* với các giá trị duyệt từ cây con bên trái của nó, sau đó mới nối với các giá trị duyệt từ cây con bên phải.
- Ở mỗi bước duyệt đệ quy qua trái hoặc phải, dùng hàm *vectorConcat* để nối các vector lại với nhau theo đúng thứ tự.

2.3. LNR:

- Sử dụng thuật toán tương tự với hàm *NLR* nhưng khác ở thứ tự gọi hàm để đệ quy.
- Ta vẫn sẽ gọi hàm *LNR* để có được các giá trị duyệt từ cây con bên trái trước, nhưng không phải nối giá trị ở *pRoot* với các giá trị trên mà sẽ nối các giá trị trên với *pRoot.* Sau đó gọi đệ quy tới cây con bên phải rồi nối tiếp các giá trị ở trên với các giá trị duyệt được bên phải.

2.4. LRN:

- Cũng tương tự thuật toán với hai hàm trên và chỉ khác ở thứ tự gọi đệ quy.
- Ta gọi lần lượt đệ quy tới cây con trái, rồi tới cây con phải, nối các giá trị trên với nhau bằng *vectorConcat*. Sau đó mới nối đến giá trị ở NODE *pRoot* đang xét.

2.5. LevelOrder:

- Sử dụng Queue để lưu trữ các NODE ở cùng một level trong cây.
- Ta bắt đầu với *pRoot* với số lượng NODE ở level này là 1, ta lưu lại ở biến *nodes* để có được số lượng NODE ở mỗi level dựa vào level trước đó. Ở mỗi lần duyệt, ta thêm 2 NODE là *p_left với p_right* của NODE đầu queue vào queue để duyệt cho level tiếp theo. Sau đó mới lấy NODE ở đầu queue ra rồi thêm giá trị của nó vào vector của từng Level.
- Sau khi duyệt xong các NODE ở cùng 1 level để có được từng vector ở từng level khác nhau, ta sử dụng vector khác để lưu trữ các vector trên.

2.6. countNode:

- Sử dụng đệ quy để đếm số NODE của NODE đang xét (tính là 1 NODE) tổng với số NODE 2 cây con bên trái và phải của nó.

2.7. sumNode:

- Sử dụng thuật toán đệ quy tương tự như *countNode* nhưng thay vì lấy tổng số NODE ở bước gọi đệ quy, ta lấy tổng giá trị của các NODE.

2.8. Level:

- Sử dụng đệ quy để tính level của một NODE *p.*
- Gọi đệ quy đến 2 nhánh của *pRoot* để tìm NODE *p,* tìm được thì sẽ trả về số NODE tính từ *pRoot* duyệt đến đó.
- Ở hai nhánh chắc chắn sẽ có một nhánh không tìm được và trả về -1, vì vậy ở kết quả cuối cùng ta cộng 2 để triệt tiêu -1 ở nhánh không tìm thấy và còn 1 để tính cho nhánh tìm thấy.

2.9. heightNode:

- Dùng hàm *findNode* để kiểm tra xem có tồn tại giá trị cần xét trong Tree hay không và tìm địa chỉ của NODE mang giá trị đó.
- Nếu có thì dùng hàm *treeHeight* để tính độ cao của NODE đó.

2.10. countLeaf:

- Dùng đệ quy đến từng nhánh của một NODE để tính số NODE lá (là NODE có nhánh trái phải đều trỏ về NULL).

II. BINARY SEARCH TREE:

1. Hàm phụ:

1.1. createNode (như BN).

1.2. findParent:

- Sử dụng đệ quy đến 2 nhánh của từng NODE để tìm NODE cha của một NODE nào đó.
- Trả về NULL nếu không có NODE cha (NODE *pRoot) hoặc không có NODE đó trong Tree.*

1.3. findLeftMost:

- Sử dụng đệ quy để tìm NODE trái nhất của một Tree nào đó.
- NODE trái nhất là NODE khi duyệt theo bên trái của một Tree nào đó thì tới NODE đó sẽ không duyệt được nữa.

1.4. findMax:

- Sử dụng đệ quy để tìm ra NODE có giá trị lớn nhất của một Tree.
- Ta gọi hàm đệ quy trước khi thực hiện thao tác để duyệt từ dưới lên, so sánh lần lượt các NODE với 2 NODE con bên trái và phải của nó để

tìm ra NODE max của các cụm NODE. Lần lượt duyệt lên thì ta sẽ có NODE max của cả Tree.

- Không sử dụng tính chất của BST nên vẫn hoạt động tốt nếu cây BST bị lỗi về mặt thứ tự.

1.5. findMin:

- Ý tưởng như hàm *findMax* nhưng để tìm NODE có giá trị nhỏ nhất của Tree.

2. Hàm chính:

2.1. Search:

- Sử dụng một con trỏ xuất phát từ *pRoot* để tìm NODE có giá trị x.
- Ta sử dụng vòng lặp với điều kiện dừng là chưa xét hết Tree và chưa xét tới NODE có giá trị x.
- Dựa vào tính chất của BST, ta điều hướng con trỏ đi từ việc so sánh x với giá trị của NODE đang xét đến.

2.2. Insert:

- Sử dụng hàm *createNode* để tạo NODE với giá trị mới.
- Nếu Tree rỗng thì ta thêm NODE đó vào Tree rồi kết thúc.
- Ngược lại, sử dụng thuật toán tương tự như *Search* để tìm vị trí thêm với 2 NODE là *pFind* và *pInsert,* với *pFind* để tìm tới vị trí thêm vào và *pInsert* để lưu lại NODE cha của vị trí thêm vào.
- So sánh giá trị cần thêm x với giá trị của NODE cha để tìm vị trí thêm phù hợp.

2.3. Remove:

- Tìm NODE cần xóa với hàm *Search*.
- Ta có 3 trường hợp của một NODE để xóa:
  + NODE có cả 2 NODE con trái và phải:
    - Ta tìm một NODE khác để thay thế vào vị trí của NODE cần xóa làm sao đảm bảo đúng tính chất của BST => Ở đây ta chọn NODE

trái nhất bên phải (NODE nhỏ nhất của cây con bên phải, sẽ là NODE liền kề với NODE cần xóa về thứ tự).

- Đổi giá trị 2 NODE trên cho nhau rồi ta đưa về trường hợp xóa NODE thay thế đó.

+ NODE chỉ có 1 NODE trái hoặc phải:

- Dùng hàm *findPar* để tìm NODE cha của NODE cần xóa, sau đó nối NODE cha vừa tìm được với NODE con của NODE cần xóa để đảm bảo thứ tự của BST. Cuối cùng xóa NODE đó ở trường hợp 3.
- Ở trường hợp xóa NODE có cả hai nhánh trái và phải, ta phải thay thế với NODE trái nhất của cây bên phải thì NODE thay thế đó vẫn có thể ở trường hợp này, vì vậy trường hợp này cũng được sử dụng để xử lý phụ cho trường hợp trên.

+ NODE lá (không có NODE trái hay phải):

- Trường hợp này dùng để xử lý cho cả 3 trường hợp (với 2 trường hợp trước đó thì dùng để xử lý giai đoạn cuối cùng là xóa NODE).
- Nếu NODE *pPar* là NODE cha của NODE cần xóa vẫn là NULL thì ta mới tiến hành tìm NODE cha của NODE cần xóa (vì 2 trường hợp trên đều cần dùng tới *pPar* nên không còn là NULL như ban đầu nữa).
- Nếu sau khi tìm *pPar* và *pPar* không là NULL, ta nối vị trí cần xóa của NODE cần xóa tại *pPar* về NULL để sau khi xóa thì Tree vẫn hợp lệ.
- Nếu *pPar* là NULL thì có 2 trường hợp:
  + NODE cần xóa là *pRoot* và cũng là NODE duy nhất của cây, ta phải cập nhật lại Tree về *NULL*.
  + NODE cần xóa là NODE đã được nối thứ tự ở trường hợp 2, không cần làm gì thêm.

- Sau khi đã thực hiện xong các thao tác chỉnh sửa cho cây, ta mới xóa NODE.

2.4. createTree:

- Tạo NODE mới cho *pRoot* của cây.
- Duyệt mảng được cung cấp rồi dùng hàm *Insert* để thêm các phần tử vào Tree một cách tự động.

2.5. removeTree:

- Sử dụng đệ quy để xóa từng NODE trong Tree, ta gọi đệ quy hàm trước khi thực hiện xóa để hàm xóa Tree từ dưới lên.
- Ở mỗi NODE ta xóa và trỏ lại về NULL, điều này giúp cập nhật lại Tree vì ta truyền Tree vào hàm theo kiểu tham chiếu.

2.6. Height:

- Tìm độ cao của cây, ý tưởng như hàm phụ *treeHeight* ở Binary Tree.

2.7. *countLess:*

- Gọi đệ quy để đếm các NODE thỏa yêu cầu ở 2 nhánh trái phải của một NODE, nếu NODE đó thỏa điều kiện thì ta tính vào và cộng số lượng lên 1.

2.8. countGreater:

- Sử dụng thuật toán tương tự hàm *countLess* và chỉ cần thay đổi điều kiện.

2.9. isBST:

- Sử dụng hai hàm *findMin* và *findMax* để tìm NODE có giá trị nhỏ nhất của bên phải và lớn nhất của bên trái.
- Sử dụng 2 NODE trên để kiểm tra điều kiện một NODE phải bé hơn toàn bộ cây con phải (phải bé hơn hoặc bằng min của cây con phải) và lớn hơn toàn bộ cây con trái (phải lớn hơn hoặc bằng max của cây con trái).
- Sử dụng đệ quy để duyệt tới các NODE dưới của Tree, trả về kết quả của 2 cây con trái phải từ việc duyệt đệ quy để kiểm tra toàn bộ Tree.

2.10.isFullBST:

- Kiểm tra ở mỗi NODE có bị khuyết NODE con ở trái hoặc phải hay không, nếu có thì trả về false cho NODE đó.
- Nếu đã đảm bảo đủ NODE con, ta sử dụng lại *isBST* để kiểm tra tính thứ tự rồi gọi đệ quy đến 2 cây con trái phải để kiểm tra toàn bộ Tree.

III. AVL TREE:

1. Hàm phụ:

1.1. isBalanced:

- Hàm kiểm tra xem một cây đã cân bằng theo tiêu chuẩn của AVL tree chưa.

1.2. findParent (như BST).

1.3. height (như BST).

1.4. heightUpdate:

- Sử dụng đệ quy để cập nhật lại chiều cao của từng NODE trong một Tree bằng hàm *height.*

1.5. RRotation:

- Để xoay một cây theo chiều phải thì ta cần sử dụng 2 NODE, hàm này sẽ xử lý theo 2 NODE với NODE được truyền vào là NODE đầu tiên của 1 NODE đó, NODE sau là NODE trái của NODE đầu tiên.
- Ta thực hiện xoay và nối giữa 2 NODE cần dùng rồi sao cho thứ tự và tiêu chuẩn của cây AVL được đảm bảo, rồi cập nhật lại *pRoot* của Tree cho NODE thứ 2, vì vậy mà ta cần truyền NODE đầu tiên theo tham chiếu.
- Việc xoay ở thuật toán này không ảnh hưởng đến NODE thứ 3 (NODE trái của NODE 2) nên ta không cần quan tâm mà chỉ cần xoay NODE thứ 2 lên rồi cập nhật NODE 1 theo NODE 2 để cân bằng Tree và đúng yêu cầu AVL Tree.

- Sau khi xoay xong thì ta cập nhật lại độ cao cho Tree vì việc xoay và nối các NODE lại có thể ảnh hưởng đến độ cao của nhiều NODE lân cận.

1.6. LRotation:

- Thuật toán tương tự như *RRotation* nhưng ta cần xoay cây qua trái nên 3 NODE cần dùng sẽ là NODE đầu tiên (NODE được truyền vào) và 2 NODE lấy qua phải của nó.

1.7. RLRotation:

- Hàm xoay Tree với trường hợp mất cân bằng theo dạng Right – Left, ta tận dụng lại 2 hàm *RRotation (quay phải) và LRotation (quay trái).*
- Ta xoay phải ở NODE phải của NODE *pRoot* rồi quay trái ở NODE *pRoot,* ta truyền trực tiếp *pRoot* và *pRoot->p_right* vào hàm để hàm tự động cập nhật sau khi xoay và cân bằng lại.

1.8. LRRotation:

- Thuật toán tương tự như *RLRotation* nhưng khác ở cách sử dụng hai hàm *RRotation* và *LRotation.*
- Ta xoay trái (*LRotation*) ở NODE thứ 2 (NODE trái của NODE đầu tiên) rồi xoay phải (*RRotation)* ở NODE *pRoot.*

1.9. balanceTree:

- Gọi đệ quy trước khi thực hiện các thao tác cân bằng để cây được cân bằng từ dưới lên.
- Sử dụng *heightUpdate* để cập nhật lại chiều cao cây trước khi kiểm tra cân bằng.
- Ta kiểm tra mỗi NODE đã được cân bằng về độ cao ở 2 cây con trái phải hay chưa, nếu chưa thì tiến hành cân bằng lại.
- Ta sử dụng con trỏ *pCheck* để kiểm tra dạng mất cân bằng bằng cách đi theo cây con có độ cao lớn hơn (vì khi mất cân bằng thì chắc chắn cây con cao hơn sẽ là cây con làm cho cây bị lệch).
- Sử dụng if-else với 4 trường hợp mất cân bằng rồi gọi hàm cân bằng tương ứng.

1.10. Search (Như BST).

1.11. FindLeftMost (Như BST).

1.12. findMax (Như BST).

1.13. findMin (Như BST).

1.14. IsBST (Như BST).

2. Hàm chính:

2.1. createNode:

- Ý tưởng như các hàm *createNode* ở BN hay BST, thêm việc cập nhật độ cao cho một NODE đơn là 0.

2.2. Insert:

- Như hàm *Insert* của BST vì cơ bản AVL Tree cũng là BST.
- Nhưng vì AVL Tree có thêm tiêu chuẩn về độ cao các cây con nên ta phải gọi hàm *balanceTree* sau khi chèn NODE mới để cân bằng lại các NODE trong Tree.

2.3. Remove:

- Xóa NODE như hàm *Remove* của BST.
- Tuy nhiên sau khi xóa NODE thì AVL Tree có thể bị mất cân bằng nên ta gọi hàm *balanceTree* để cân bằng lại cây.

2.4. isAVL:

- Kiểm tra 2 tính chất của cây AVL: phải là cây BST và phải được cân bằng về độ cao (dùng hàm *IsBalanced* và *isBST*).
- Gọi đệ quy để kiểm tra đến các NODE tiếp theo ở 2 cây con trái phải của Tree.

MINH CHỨNG UP FILE SOURCE CODE LÊN GITHUB: