Caijun Qin
02/09/2020

PA1 Part B

a) What methods will you need to add? Write pseudocode for these methods.

Several new methods will be added to ensure balance in the BST. Assume that previous methods in the basic BST tree are still available.

A basic helper method to find the height of a tree or subtree given a starting node will be used in other functions, especially a balance factor determining function. Height here is defined as the edge length of the longest path from the given root to a leaf.

```
int getHeight(const node &start){
    //edge case: empty tree, meaning start is not part of this tree
    if(empty()) { return 0; }

    //edge case: start is the only node in tree or subtree
    if(countChildren(start) == 0) { return 0; }

    node * ptr = &start;
    if(ptr == nullptr) {return 0;}
    return 1 + max(getHeight(ptr -> left_child), getHeight(ptr -> right_child));
}
```

The function below determines the balance factor at the root of the BST or a subtree. This is useful if nodes are not allowed to carry extra information such as storing balance factor.

```
int balanceFactor(const node &start){
    //edge case: empty tree, meaning start is not part of this tree
    if(empty()) { throw std::out_of_range("BST is empty!"); }

    node * ptr = &start;
    return getHeight(ptr -> right_child) – getHeight(ptr -> left_child);
}
```

Four (4) different rotation methods are needed in case the insertion of a new node makes the AVL tree unbalanced. These methods are helper functions to be used inside an overridden insert function for AVL trees. Again, let *start* be the root of the tree or a subtree.

```
void rotateLeft(const node &start){
    //edge case: empty tree, meaning start is not part of this tree
    if(empty()) { { throw std::out_of_range("BST is empty!"); }

    node * g = &start; //grandparent of inserted node
    //prerequisite: must have right child
    if(g -> right_child == nullptr) { throw std::out_of_range("Missing right child!");
```

Caijun Qin
02/09/2020

```
        node * p = g -> right_child;
        g -> right_child = p -> left_child;
        p -> left_child = g; //parent of inserted node
}

void rotateRight(const node &start){
        //edge case: empty tree, meaning start is not part of this tree
        if(empty()) { { throw std::out_of_range(“BST is empty!”); }

        node * g = &start; //grandparent of inserted node
        //prerequisite: must have left child
        if(g -> left_child == nullptr) { throw std::out_of_range(“Missing left child!”);

        node * p = g -> left_child; //parent of inserted node
        g -> left_child = p -> right_child;
        p -> right_child = g;
}

void rotateLeftRight(const node &start){
        //edge case: empty tree, meaning start is not part of this tree
        if(empty()) { { throw std::out_of_range(“BST is empty!”); }

        node * g = &start; //grandparent of inserted node
        //prerequisite: must have left child
        if(g -> left_child == nullptr) { throw std::out_of_range(“Missing left child!”);

        node * p = g -> left_child; //parent of inserted node
        rotateLeft(*p);
        rotateRight(*g);
}

void rotateRightLeft(const node &start){
        //edge case: empty tree, meaning start is not part of this tree
        if(empty()) { { throw std::out_of_range(“BST is empty!”); }

        node * g = &start; //grandparent of inserted node
        //prerequisite: must have left child
        if(g -> right_child == nullptr) { throw std::out_of_range(“Missing right child!”);

        node * p = g -> right_child; //parent of inserted node
        rotateRight(*p);
        rotateLeft (*g);
}
```

Caijun Qin
02/09/2020

b) How will you need to change your tree data structure to account for these new methods?

Structurally, the tree will be mostly the same as it is still a binary tree. However, there will be major changes to many mutator functions. If AVL tree is another class that inherits from BST, then insert and erase will be overriden and contain extra code for balancing the tree. In insert, as previously mentioned, a new node will still be inserted using standard BST insert. Afterwards, the balance factor will be recalculated for ancestor nodes from the inserted node upwards until the first occurrence of a balance factor outside the [-1, 1] range is found. Then, the appropriate rotation method is called. This decision is based on a temporary boolean array that records the directions of the most recent 2 movements when traversing the ancestral path. For example, {false, false} indicates left-left, so rotateLeft will be called on the first unbalanced ancestral node. For erase, standard BST deletion will be performed first. Adding on to the overriden erase function, the same process of recalculating balance factors upwards for ancestral nodes is done until the first unbalanced node is found. Then, the appropriate rotation function is called on that node. If the node struct has a field to store height, then inserting or deleting a node simply propagates an increment or decrement, respectively, up the ancestral path of nodes. At each update on height, a node's rebalancing factor can be computed much faster than with a recursive height function.