CSC 3100 Midterm Report

O(logN) Insertion, Deletion, and Median Based on AVL Tree

Wei Wu (吴畏)

118010335

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

1. Introduction

According to the requirement, operations including insertion, deletion, and median should have time complexity strictly less than O(N). After considering array, linked list, skip list, and heap, we decided to use a balanced binary search tree, where insertion, deletion are O(logN).

According to the textbook, a binary search tree is a binary tree where left subtree < parent < right subtree holds for every internal node. By this definition, searching for a number takes O(H) time, where H is the height of the tree. If the tree is full, there will be logN levels. However, if not full, there can be at most N levels. Then, the above algorithms will be O(N), which does not satisfy the requirement.

Therefore, we need to keep the tree balanced, in other words, nearly full. After reading the textbook, we found the idea of AVL Tree, where the heights of leaves are equal or differ by 1. The definition of AVL Tree guarantees that searching can be done in O(logN). On the other hand, we must keep the binary search tree balanced after each insertion and deletion, which is the most challenging part.

2. Insertion

To insert a number, we need to know the location to insert first. The location (parent node to attach to) can be determined by searching for the number. Assume that the tree is balanced at first. Searching takes O(logN) time since we can eliminate about half of the remaining nodes. Note that, in the original AVL Tree, inserting a duplicated number is not allowed. Therefore, we can add a data field called count to each node to represent the frequency of that number. When the number to insert already exists in the tree, we can simply increase count by 1. If the number does not exist, we can create a new node and insert the new node by attaching it to the parent, which can be done in O(1) time.

After the insertion, the tree could be unbalanced. Since the tree is balanced before the insertion, the heights of subtrees can differ by at most 2. If unbalanced, there exists a node X whose one subtree is of height H and other subtree is of height H+2.

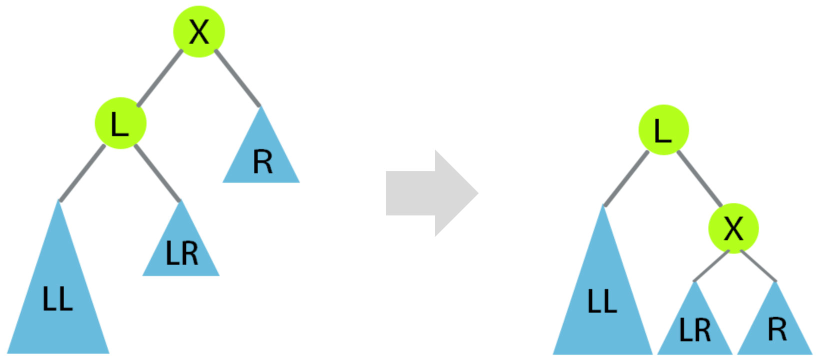

Figure 1. ***Unbalanced Case 1 (Left Left & Right Right)***

The first case is that the new node is attached to the left subtree of the left child of X or the right subtree of the right child of X (see the tree on the left). Consider the left-left case. Observe that LL < L < LR < X < R. Hence, we can pick L as the new root and rotate the whole subtree (see the tree on the right). Note that the new subtree is an AVL tree.

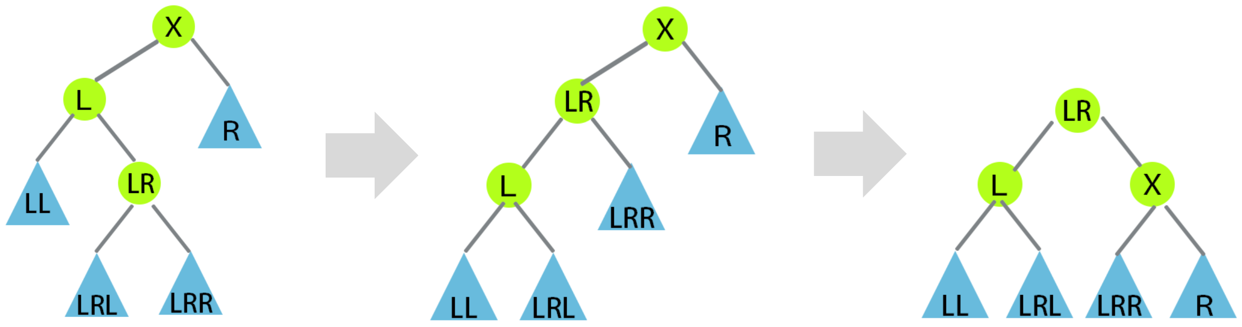

Figure 2. ***Unbalanced Case 2 (Left Right & Right Left)***

The second case is that the new node is attached to the right subtree of the left child of X or the left subtree of the right child of X (see the tree on the left). Consider the left-right case. Observe that LL < L < LRL < LR < LRR < X < R. First, we rotate the left subtree of X by picking LR as the new root (see the tree in the middle). Then, pick LR as the new root and rotate the whole subtree (see the tree on the right).

The above rotation can be done in O(1) time, since the number of operations are fixed. Therefore,

T(insertion) = T(searching) + T(creating & attaching) + T(rebalancing)
= O(logN) + O(1) + O(1)
= O(logN)

```
Pseudo-code
Node* insertNode (Node* node, int key)
    // Step 1: Locate and insert the node
    if (node == NULL) return(createNode(key));
    if (key < node->key)
        node->left = insertNode(node->left, key);
        node->lsize++;
    else if (key > node->key)
        node->right = insertNode(node->right, key);
        node->rsize++;
    else
        node->count++;
    // Step 2: Update the height
    node->height = max(height(node->left), height(node->right)) + 1;
    // Step 3: Calculate the balance factor
    int BF = balanceFactor(node);
    if (BF > 1 && key < node->left->key)
        return rotateRight(node);
    else if (BF < -1 && key > node->right->key)
        return rotateLeft(node);
    else if (BF > 1 && key > node->left->key)
        node->left = rotateLeft(node->left);
        return rotateRight(node);
    else if (BF < -1 && key < node->right->key)
        node->right = rotateRight(node->right);
        return rotateLeft(node);
    return node;
```

3. Deletion

Similar to insertion, we need to search for the node to delete first, which takes O(logN) time. If not found, we do nothing. If found, and count of the node is greater than 1, we can simply decrease count by 1 and return. Otherwise, if count is 1, there are three situations. First, if the node is a leaf, we delete it directly. Second, if the node has only 1 child, we swap it with its child and then delete the child. Third, if the node has 2 children, we first find its predecessor (the greatest node in the subtree that is smaller than the node to delete). This takes O(logN) time. Then, we swap it with its predecessor. If the predecessor has no left child, we delete the predecessor directly. Otherwise, we swap the predecessor with its left child, and then delete the left child. Then, we record the location of deletion. In summary, this procedure (deattaching and deleting) takes O(logN) time.

After the deletion, the tree might be unbalanced. If so, we need to rearrange the tree. Similar to part

2, there exists an internal node whose one subtree is of height H and other subtree is of height H+2. To identify such a node, we define a variable called balance factor BF with respect to each node. BF = the height of the left subtree – the height of the right subtree. If BF = -1, 0, or 1, the subtree is balanced.

After deletion, we calculate and check BF from the location of deletion to the root. That is, from bottom to top. If BF = -2 or 2, we need to rotate the subtree. The situations and methods to perform rotation are similar to those in part 2. Thus, a rotation takes O(1) time. However, after one rotation, the total height of the subtree might change by 1. In that case, we need to further check whether the upper subtree is unbalanced. If so, we perform another rotation. We should repeat the above procedures until the current subtree is balanced or the root of the tree is reached. So, rebalance takes O(logN). Therefore,

T(deletion) = T(searching) + T(deattaching and deleting) + T(rebalancing)
= O(logN) + O(logN) + O(logN)
= O(logN)

```
Pseudo-code
Node* deleteNode (Node* root, int key, bool exist = false)
    // Step 0: Check if the key exists
    if (!exist)
        if (!search(root, key))
            return root;
    // Step 1: Locate and delete the node
    if (root == NULL)
        return root;
    if (key < root->key)
        root->left = deleteNode(root->left, key, true);
        root->lsize--;
    else if(key > root->key)
        root->right = deleteNode(root->right, key, true);
        root->rsize--;
    else
        if (root->count > 1)
            root->count--;
            return root;
        If (root->left == NULL && root->right == NULL)
                delete root;
                return NULL
        else if (root->left == NULL || root->right == NULL)
            Node *node;
            if (root->left)
                node = root->left;
            else
                node = root->right;
            *root = *node;
            delete node;
        else
            Node* node;
            Node* parent = maxValueNodeParent(root);
            if (parent == root) node = root->left;
            else node = parent->right;
            root->key = node->key;
```

```
          root->lsize--;
          if (node->left)
             *node = *node->left;
             delete node->left;
          else
             if (parent == root)
                root->left = NULL;
                delete node;
             else
                parent->right = NULL;
                delete node;
    // Step 2: Update the height
    root->height = max(height(root->left), height(root->right)) + 1;
    // Step 3: Calculate the balance factor
    int BF = balanceFactor(root);
    if (BF > 1 && balanceFactor(root->left) >= 0)
       return rotateRight(root);
    if (BF < -1 && balanceFactor(root->right) <= 0)
       return rotateLeft(root);
    if (BF > 1 && balanceFactor(root->left) < 0)
       root->left = rotateLeft(root->left);
       return rotateRight(root);
    if (BF < -1 && balanceFactor (root->right) > 0)
       root->right = rotateRight(root->right);
       return rotateLeft(root);
    return root;
```

4. Median

Finding the median is the special case of finding the kth largest number. In an AVL tree, we know that left subtree < parent < right subtree. If we add two additional data fields, the size of the left subtree (lsize) and the size of the right subtree (rsize), to each node, we can find the kth largest number in the following binary-search-like way:

```
int kth(node* root, int k)
    if (root == NULL) return 0;
    if (k <= root->rsize) return kth(root->right, k);
   else if (root->rsize+1 <= k && k <= root->rsize+root->count)
       return root->key;
   else return kth(root->left, k-root->rsize-root->count);

int median(Node* root)
    if (root == NULL) return 0;
   return kth(root, (root->lsize + root->rsize + root->count+1)/2);
```

T(median) = T(kth largest number) = O(logN).

Meanwhile, when we do insertion and deletion (inserting a node, deleting a node, and rotating a subtree), we should keep track of lsize and rsize. Each such operation only takes O(1) time. Hence, the median algorithm does not affect the time complexity of insertion and deletion.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms third edition. MIT press.

FifteenthOfJuly. (2019, April 16). The insertion and deletion of AVL Tree (detailed explanation). Retrieved July 04, 2020, from https://blog.csdn.net/qq_29590355/article/details/89324655

GeeksforGeeks. (2019, August 07). AVL Tree: set 1 (insertion). Retrieved July 04, 2020, from https://www.geeksforgeeks.org/avl-tree-set-1-insertion/

GeeksforGeeks. (2019, August 07). AVL Tree: set 2 (deletion). Retrieved July 04, 2020, from https://www.geeksforgeeks.org/avl-tree-set-2-deletion/

Zhangbaochong. (2016, January 28). Detailed explanation of balanced binary tree. Retrieved July 04, 2020, from https://www.cnblogs.com/zhangbaochong/p/5164994.html

Classmate: Yu Mao (毛宇)