1/5/2023

SWE201

Data Structures And Algorithms

Lecturer: Doç. Dr. Devrim AKGÜN

Merve KILCI
B211202375
SOFTWARE ENGINEERING/2ND CLASS

AVL tree can be defined as height balanced binary search tree in which each node associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

There is said to be balanced if balance factor of each node is in between -1 to 1 otherwise the tree will be unbalanced and need to be balanced.

Balanced factor(k)= height(left(k))- height(right(k))

COMPLEXITY OF AVL TREE

Algorithm	Average Case	Worst Case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

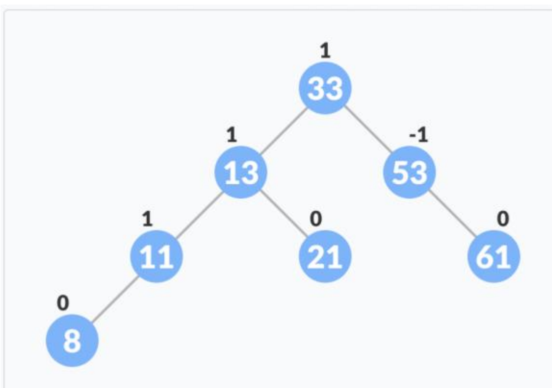
- Algorithm to insert a newNode
- And finding their height

```
// AVL tree implementation in C++

#include <iostream>
using namespace std;

class Node {
   public:
  int key;
  Node *left;
  Node *right;
  int height;
};

int max(int a, int b);

// Calculate height
int height(Node *N) {
  if (N == NULL)
    return 0;
  return N->height;
}

int max(int a, int b) {
```

```
    return (a > b) ? a : b;
  }

  // New node creation
  Node *newNode(int key) {
    Node *node = new Node();
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;
    return (node);
  }
```

A newNode is always inserted as a leaf node with balance factor equal to 0.

1.Let the initial tree be:



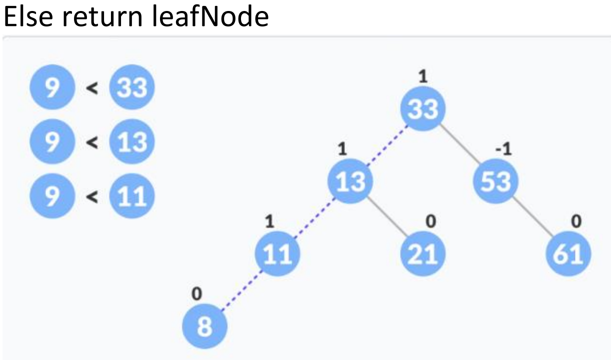
Initial tree for insertion

2.Let node to inserted to be 9

Go to the appropriate leaf node to insert a newNode using to following recursive steps.

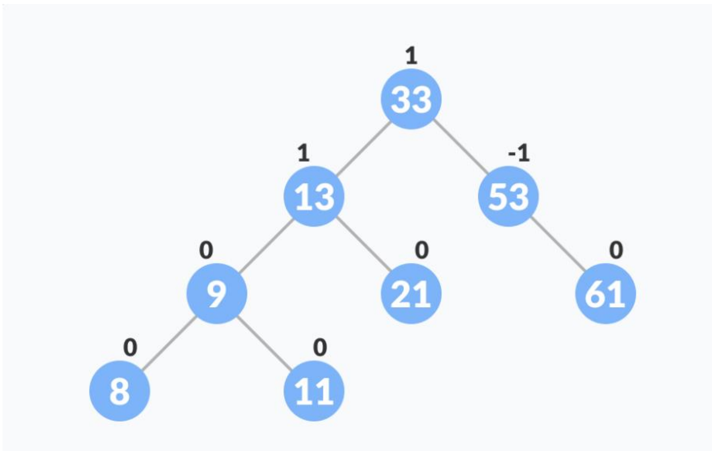
Compare newKey<rootKey and newKey>rootKey

Else return leafNode



Finding the location to insert newNode

Compare newKey <leafKey
Update balanceFactor of the nodes.
If the nodes are unbalanced then rebalance the node
The final tree is:



Final balanced tree

```
// Insert a node
Node *insertNode(Node *node, int key) {
  // Find the correct postion and insert the node
  if (node == NULL)
    return (newNode(key));
  if (key < node->key)
    node->left = insertNode(node->left, key);
  else if (key > node->key)
    node->right = insertNode(node->right, key);
  else
    return node;

  // Update the balance factor of each node and
  // balance the tree
  node->height = 1 + max(height(node->left),
            height(node->right));
  int balanceFactor = getBalanceFactor(node);
  if (balanceFactor > 1) {
    if (key < node->left->key) {
      return rightRotate(node);
    } else if (key > node->left->key) {
      node->left = leftRotate(node->left);
      return rightRotate(node);
```

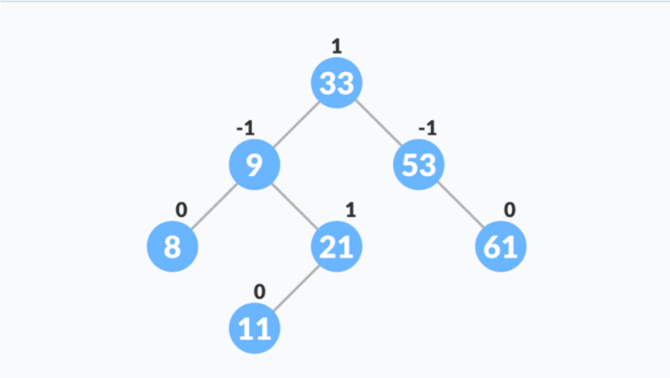
```
    }
  }
  if (balanceFactor < -1) {
    if (key > node->right->key) {
      return leftRotate(node);
    } else if (key < node->right->key) {
      node->right = rightRotate(node->right);
      return leftRotate(node);
    }
  }
  return node;
}
```

```
int main() {
  Node *root = NULL;
  root = insertNode(root, 33);
  root = insertNode(root, 13);
  root = insertNode(root, 53);
  root = insertNode(root, 9);
  root = insertNode(root, 21);
  root = insertNode(root, 61);
  root = insertNode(root, 8);
  root = insertNode(root, 11);
  printTree(root, "", true);
  root = deleteNode(root, 13);
  cout << "After deleting " << endl;
  printTree(root, "", true);
}
```

Algorithm to delete a node:

A node is always deleted as a leaf node. After deleting a node, the balance factors of the nodes get changed. In order to rebalance the factor, suitable rotations are performed.

1.Locate nodeToBeDeleted
Look if the nodeToBeDeleted is a leaf node or has one child or has two children,
Substitute the contents of nodeToBeDeleted with that of w,
Remove the leaf node w,
Update the balanceFactor of the nodes,
Rebalance the tree if the balance factor of any of the nodes is not equal to -1,0,1.
The final tree is:



Avl tree final

```
// Delete a node
Node *deleteNode(Node *root, int key) {
 // Find the node and delete it
 if (root == NULL)
  return root;
 if (key < root->key)
  root->left = deleteNode(root->left, key);
 else if (key > root->key)
  root->right = deleteNode(root->right, key);
 else {
  if ((root->left == NULL) ||
   (root->right == NULL)) {
   Node *temp = root->left ? root->left : root->right;
   if (temp == NULL) {
    temp = root;
    root = NULL;
   } else
    *root = *temp;
   free(temp);
  } else {
   Node *temp = nodeWithMimumValue(root->right);
   root->key = temp->key;
   root->right = deleteNode(root->right,
          temp->key);
  }
 }

 if (root == NULL)
  return root;
```

```
// Function to find a key in AVL tree
bool AVLsearch(
    struct Node* root, int key)
{
    // If root is NULL
    if (root == NULL)
        return false;

    // If found, return true
    else if (root->key == key)
        return true;

    // Recur to the left subtree if
    // the current node's value is
    // greater than key
    else if (root->key > key) {
        bool val = AVLsearch(root->left, key);
        return val;
    }

    // Otherwise, recur to the
    // right subtree
    else {
        bool val = AVLsearch(root->right, key);
        return val;
    }
}


// Function call to search for a node
    bool found = AVLsearch(root, 40);
    if (found)
        cout << "value found";
    else
        cout << "value not found";
```

OUTPUT OF THE PROGRAM:

```
R----33
   L----13
   |  L----9
   |  |  L----8
   |  |  R----11
   |  R----21
   R----53
      R----61
After deleting
R----33
   L----9
   |  L----8
   |  R----21
   |     L----11
   R----53
      R----61
```

//And for the search operation it will write on the output page:

```
value not found
```