**AVL Group assignment**

Populate a tree via a text file (input.txt) Make sure that after every insert, the tree is balanced. At the end, display the tree in level format. Make sure to include the height and the balance factor of every node in your output. Redirect the display to an output file (output.txt)

Hint:

//I will not accept any other algorithm

//This is not a recursive algorithm

node \* rebalance(node \*node){

node->height = max(height(node->left), height(node->right)) + 1;

int balance = getBalance(node); //node->left - node->right

/\*do rotations as necessary

If Left heavy outside : return rightRotate(node);

If right heavy outside: return leftRotate(node);

If left heavy inside: left rotation first, right rotation 2nd, return top node

node->left = leftRotate(node->left);

return rightRotate(node);

if right heavy inside: right rotation first, left rotation 2nd, return top node

node->right = rightRotate(node->right);

return leftRotate(node);

if no rotation, return node

\*/

}

//non-tail recursive algorithm because of rebalance

node\* insert(node\* node, int key)

{

//recursive Code for inserting a node

//When insert happens set height to 0 for the node

if (node == NULL)

return(newNode(key));

if (key < node->key)

node->left = insert(node->left, key);

else

node->right = insert(node->right, key);

node=rebalance(node); //update heights and rebalance

return node;

}

node \*leftRotate(node \*x){

struct node \*y=x->right;

//add more code to rotate to the left, update heights for x and y

//return y

}

node \*rightRotate(node \*x){

struct node \*y=x->left;

//add more code to rotate to the right, update heights for x and y

//return y

}

AVL Assignment

1. How do I populate the tree?

A: You will use a text file that contains a series of numbers separated by some kind of white space. The numbers can appear in any random order. You can be guaranteed that they will all be integers and there is no error checking that needs to be done.

1. Can I use my own algorithm for balancing the tree?

A: You must use the algorithm that I have provided

1. How do I display the tree in print level format

A: In the lecture notes, you will find a document called print level. There are 2 algorithms that are available to you. One uses 2 queues and the other uses only one queue. You can use either algorithm to display the tree level by level. Please note that you don’t have to worry about the children lining up with the parents. Each level will be on its own separate line. The numbers can be separated by spaces. As long as the numbers are sitting on the proper level, you are good.

1. How are the heights being updated

A: You will insert a node at the leaf position with the height of 0 and will climb up to connect the newly created node to a parent node. During the climb up, you call the balance function which will update the height. Given this height, we calculate the new balance and check to find out if we have to rotate. Inside the rotate function, we update the height again since nodes are moving to different places. We return back to the balance function, which will in turn go back to the insert function, which then continues to climb back up. This process is continued until it reaches the root.

So the quick answer is the balance function along with the rotate functions both update the heights. When the new node is created at the leaf position, its height is automatically assigned to 0



**(Using two queues)**

|  |  |  |
| --- | --- | --- |
| Current level | Next level |  |
| A | B C | Display A, enqueue its children, dequeue A,  point currentlevel to Next level, Point next level to Null |
| B C | D E F G | Display B, enqueue its children, dequeue B,  Display C, enqueue its children, dequeue C,  point currentlevel to Next level, Point next level to Null |
| D E F G | H | Display D, no children,dequeue D  Display E, no childen, dequeue E  Display F, no children, dequeue F  Display G, enqueue child, dequeue G  point currentlevel to Next level, Point next level to Null |
| H |  | Display H, no children, dequeue H  point currentlevel to Next level which is null. Next level is already null |

Enqueue root on CurrentLevel //1st time it would be A

While currentLevel

Display currentlevel //1st time (A)

Dequeue currentLevel // 1st time dequeue (A), now empty

Enqueue left nextLevel //1st time enqueue (B)

Enqueue right nextLevel //1st time enqueue (C)

If currentlevel==NULL //1st time it is null

Currentlevel=nextLevel //1st time points to B C

Nextlevel=null //Points to nothing

cout<<endl //go to next line

**(Using a single queue)**

|  |  |  |  |
| --- | --- | --- | --- |
| queue | Nodes in Current level | Nodes in next level |  |
| A | 1 | 2 | Display A, enqueue its children,increment nextLevel, dequeue A, decrement currentLevel  Currentlevel=Next level, next level=0 |
| B C | 2 | 4 | Display B, enqueue its children, increment nextLevel dequeue B, decrement currentLevel  Display C, enqueue its children, increment nextLevel dequeue C, decrement currentLevel  Currentlevel=Next level, next level=0 |
| D E F G | 4 | 1 | Display D, no children, dequeue D, decrement currentLevel  Display E, no children, dequeue E, decrement currentLevel  Display F, no children, dequeue F, decrement currentLevel  Display G, enqueue its children, increment nextLevel, dequeue G, decrement currentLevel  Currentlevel=Next level, next level=0 |
| H | 1 | 0 | Display H, no children, dequeue F, decrement currentLevel  Currentlevel=Next level which is 0,  next level=0 (DONE) |

Enqueue root ( NodesOnCurrentLevel++) // add node in current level

While NodesinCurrentLevel!=0 //or nodesInCurrentLevel !=0

Dequeue and display //1st time (A)

nodesInCurrentLevel-- //1st time dequeue A,NICL=0

enqueue left child( nodesInNextLevel++) //1st time enqueue B

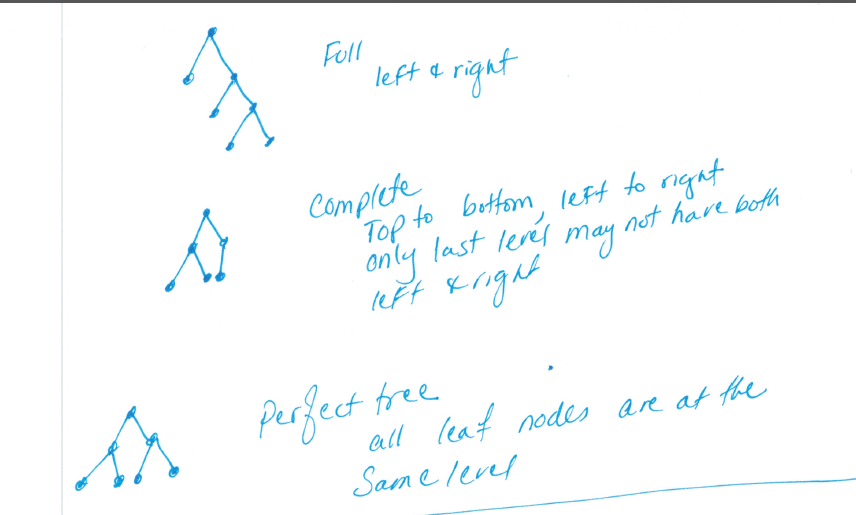
enqueue right child ( nodesInNextLevel++) //1st time enqueue C

If nodesInCurrentLevel==0 //1st time NICL=0

NodesInCurrentlevel=NodesInNextLevel //points to B C

NodeinNextlevel=0 //Points to nothing

cout<<endl //go to next line



The **height** of a node is the length of the longest downward path to a leaf from that node. The height of the root is the **height of the tree**.

The **depth** of a node is the length of the path to its root (i.e., its root path). This is commonly needed in the manipulation of the various self-balancing trees, AVL Trees in particular. The root node has depth zero, leaf nodes have height zero, and a tree with only a single node (hence both a root and leaf) has depth and height zero. Conventionally, an empty tree (tree with no nodes, if such are allowed) has depth and height −1.

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |



struct BinaryTree {

int item;

int height;

BinaryTree \*left;

BinaryTree \*right;

};

|  |  |
| --- | --- |
| Node address | |
| \*left | Item | | height | | \*right |

|  |  |
| --- | --- |
| Node address | |
| \*left | Item | | height | | \*right |

|  |  |
| --- | --- |
| Node address | |
| \*left | Item | | height | | \*right |

An AVL tree (Georgy Adelson-Velsky and Landis' tree, named after the inventors) is a [self-balancing binary search tree](http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree). In an AVL tree, the [heights](http://en.wikipedia.org/wiki/Tree_height) of the two [child](http://en.wikipedia.org/wiki/Child_node) subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. The benefit of AVL trees over Binary Search Trees is that the number of comparisons required, i.e. the AVL tree's height, is guaranteed never to exceed log(n).

An AVL tree is a binary search tree which has the following properties:

1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.

Trees are balanced through rotations. Tree rotation is an operation on a [binary tree](http://en.wikipedia.org/wiki/Binary_tree) that changes the structure without interfering with the order of the elements. A tree rotation moves one node up in the tree and one node down. It is used to change the shape of the tree, and in particular to decrease its height by moving smaller subtrees down and larger subtrees up, resulting in improved performance of many tree operations. When a subtree is rotated, the subtree side upon which it is rotated increases its height by one node while the other subtree decreases its height. This makes tree rotations useful for rebalancing a tree

If positive number, it is left heavy, if negative number it is right heavy.

if balance >1 or balance <-1 then rotate.

If >1 go the left child (left heavy)

If child is negative (right heavy)

(right heavy, inside heavy) double rotate(Left rotate, right rotate)

else (left heavy)

right rotate

If <-1 go to the right child (right heavy)

If child is positive, (left heavy)

(left heavy, inside heavy) double rotate (right rotate, left rotate)

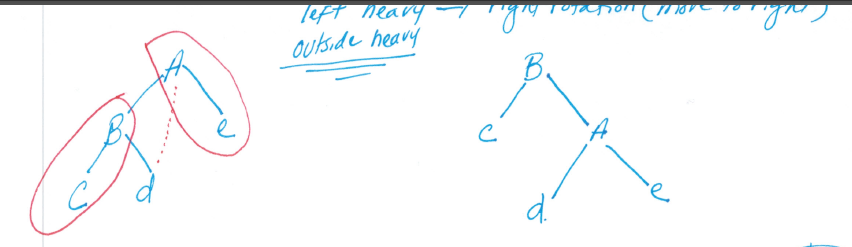
else

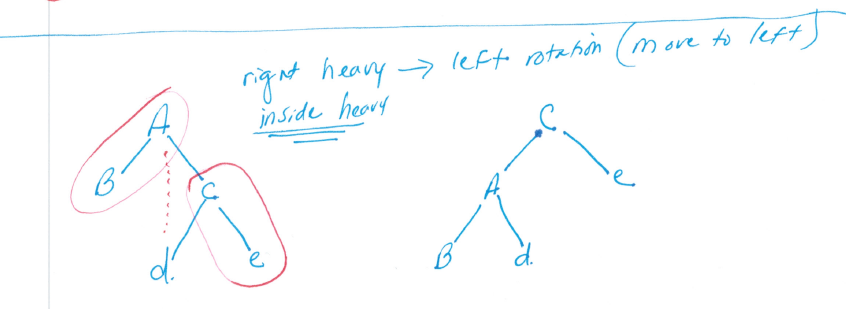
left rotate (right heavy)

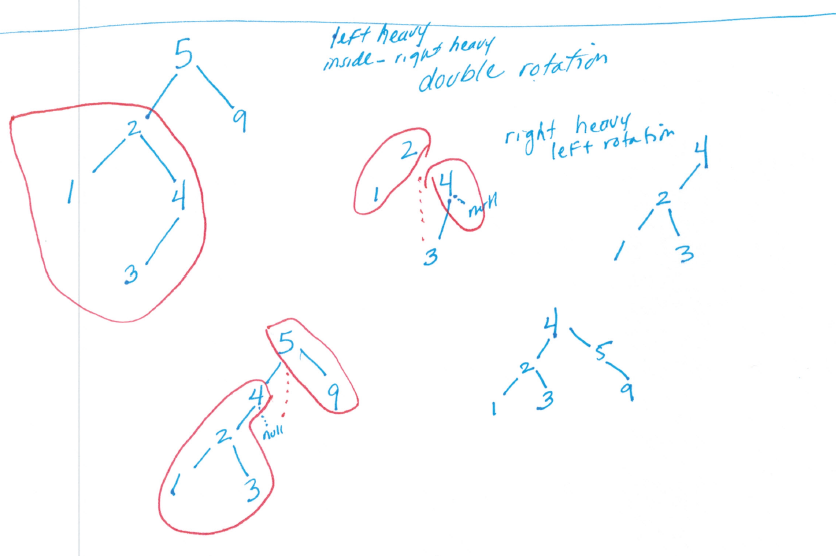
If outside heavy on the left, rotate to the right (balance is positive)

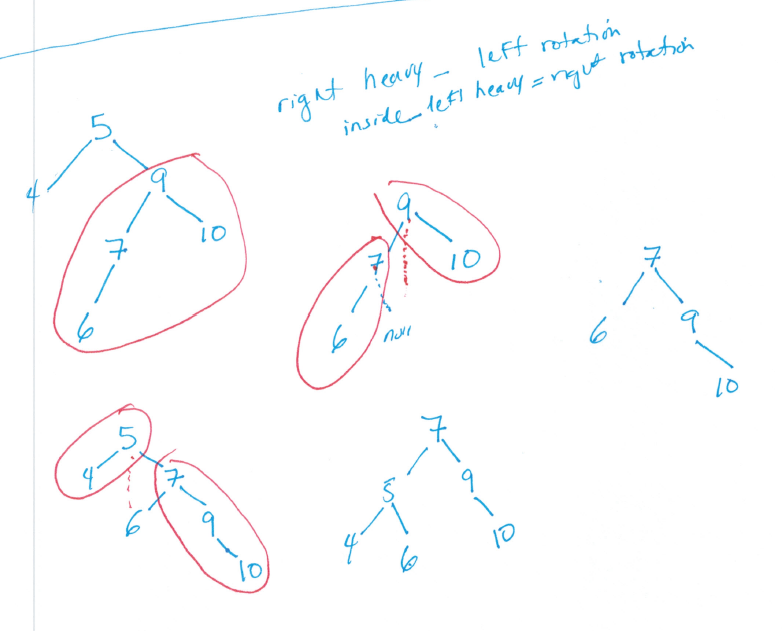
If outside heavy on the right, rotate to the left (balance is negative)

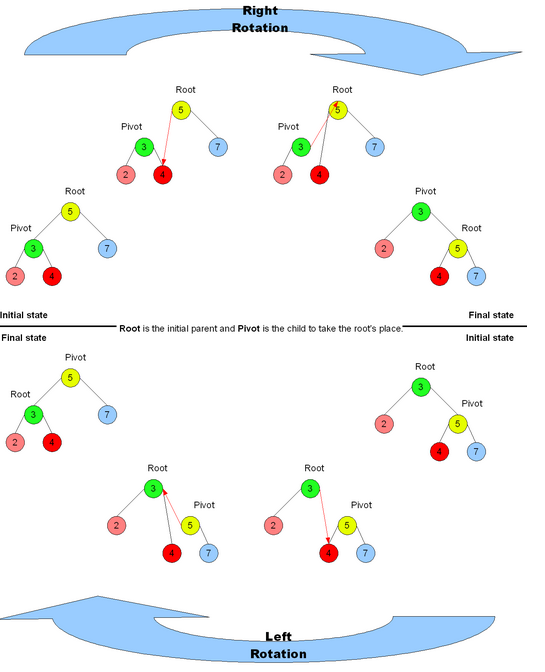
|  |  |
| --- | --- |
| Left heavy  (Outside)  Right heavy  (outside)  Left heavy  Right heavy  (inside)  Right heavy  Left heavy  (inside) |  |











//This is not a recursive algorithm

struct node \* rebalance(struct node \*node){

node->height = max(node->left->height, node->right->height) + 1;

int balance = getBalance(node); //node->left - node->right

/\*

rotation conditions based on balance

node right heavy <-1

right child is left heavy >0

//double rotation right, left case #3

node->right=rotate right on child

return rotate left on node

else

return rotate left on node //single left rotation case #2

node is left heavy> 1

left child is right heavy < 0

//double rotation left, right case #4

node->left=rotate left on child

return rotate right on node

else

return rotate right on node //single right rotation case #1

(The first arg in parenthesis deals with the node whose balance is > 1 or < -1)

(The second arg in parenthesis deals with the node under it and its balance)

(+,+) // 1st is left heavy, 2nd it is outside heavy (single right rotation

(-,-) // 1st is right heavy, 2nd is outside heavy (single left rotation)

(+,-) // 1st is left heavy, 2nd is inside heavy (double rotation, left and right)

(-,+) // 1st is right heavy, 2nd is inside heavy(double rotation, right, and left)

The numbers in paranthesis are the balance factors

case 1 ============== single right rotation + , +

5 (2) 3

3(1) 1 5

1

case 2 =============== left rotation - , -

5(-2) 6

6 (-1) 5 7

7

case 3 =============== right (inside), left (outside) rotation (Double)

5 (-2) 7 7

8 (1) 8 5 8

7

case 4 ============= left (inside), right (outside) rotation (Double)

5(2) 3 3

2(-1) 2 2 5

3

\*/

}

//non-tail recursive algorithm because of rebalance

struct node\* insert(struct node\* node, int key)

{

//recursive Code for inserting a node

//When insert happens set height to 0 for the node

if (node == NULL)

return(newNode(key));

if (key < node->key)

node->left = insert(node->left, key);

else

node->right = insert(node->right, key);

node=rebalance(node); //update heights and rebalance

return node;

}

struct node \*leftRotate(struct node \*x){

struct node \*y=x->right;

//add more code to rotate to the left, update heights for x and y

//return root of the tree

}

struct node \*rightRotate(struct node \*x){

struct node \*y=x->left;

//add more code to rotate to the right, update heights for x and y

//return root of the tree

}

Insert at the leaf. The inserted node will have a height of 0.

Since inserting occurred by traversing from the root all the way to a leaf node, recursively, we have to go back until we end up back at the root. As we go back, rebalance will be called on the nodes along the path. This causes the height of the node to be updated since we just added a node. The balance factor is examined for the given node and if it is >1 or <-1, then rotation occurs.

When rotation occurs, the location of the node on which the rotation occurs will change. This means that this change has to be reflected back to the insert so that the right or the left members are updated.

Example: Inserts into an AVL tree

Insert (NULL, 9)

Insert (9,4)

Insert (9,10)

Insert (9,2)

Insert (9,3)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ins  9 | |  | | --- | | Insert (NULL, 9) NULL  Return(9) //root | | 9 (0,0) //only one node. Rebalance is not called  Note: 1st number is height, 2nd number is balance. Only height is stored. | |
| ins  4 | |  | | --- | | Insert (9,4) 9  Left=insert(NULL, 4)  Node= rebalance (9)  Return 9 //root | | Insert (NULL, 4)  Create node  Return (4) |   Inserts at leaf node.  9 (0,0) //needs updating  4 (0,0) | |  | | --- | | rebalance (9) 9  H9=Max(H4,NULL)+1  Balance=0 - -1 (1)  Return 9 | | 9 (1,1) //updated  4 (0,0) |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ins  10 | |  | | --- | | Insert (9,10) 9  right=insert(NULL, 10)  Node= rebalance (9)  Return 9 //root | | insert (NULL,10)  Create node  return (10) |   Inserts at leaf node.  9 (1,0) //needs updating  4 (0,0) 10 (0,0) | |  | | --- | | rebalance (9) 9  H9=Max(H4,H10)+1  Balance=0-0  Return 9 | | 9 (1,0) //stays the same  4(0,0) 10 (0,0) |
| ins  2 | |  | | --- | | Insert (9,2) 9  left=insert(4, 2)  Node= rebalance (9)  Return 9 //root | | Insert (4, 2) 4  Left=insert (NULL,2)  Node= rebalance (4)  Return 4 | | insert (NULL,2)  create node  return (2) |   9(1,0) //needs update  4(0,0) 10(0,0) //needs update  2(0,0) | |  | | --- | | rebalance (9) 9  H9=max(H4,H10)+1  Balance=1-0  Return 9 |  |  | | --- | | rebalance (4) 4  H4=Max(H2,NULL)+1  Balance=0 - -1  Return 4 | | ================================  9(2,1) //update 9  4(1,1) 10(0,0)  2(0,0)  ==============================  9(1,0)  4(1,1) 10(0,0) //update 4  2(0,0) |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ins  3 | |  | | --- | | 1  Insert (9,3) 9  left=insert(4, 3)  Node= rebalance (9)  Return 9 //root | | 2  Insert (4, 3) 3  Left=insert (2,3)  Node= rebalance (4)  Return Node (3)  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  NOTE what is being returned | | 3  insert (2,3) 2  right=insert(NULL, 3)  Node= rebalance (2)  Return (2) | | 4  Insert (NULL, 3)  Create node  Return 3 |   This is what happens after 3 has been inserted. The height is automatically set to 0.  9(2,1)  4(1,1) 10 (0,0)  2(0,0)  3(0,0) | |  | | --- | | 7  rebalance (9) 9  H9=max(H3,H10)+1  Balance=2-1  Return 9 | | 6  rebalance (4) 4  H4=max(H2,NULL)+1  Balance=1- -1 (2)  Left= RotLeft(2) 3  Return RotRightt(4) 3 | | 5  rebalance (2) 2  H2=max(NULL,H3)+1  Balance=-1-0  Return 2 |   The heights are updated for 2, 4, 9. The balances are checked. If it is >1 or <-1 then rotation is needed. | ==================    9 (2,1) //update  3(1,0) 10 (0,0)  2(0,0) 4(0,0)  ================  9  4(2,2) 10  2(1,-1)  3(0,0)  =================  9  4 10  2(1, -1) //updated  3(0,0) | double rotation  Rotate right on 2  Rotate left on 4  Given the following: first rotate left on 2  9  4(2,2) 10  2(1,-1)  3(0,0)  The result of left rotation is  9  4(2,2) 10  3(1,1)  2(0,0)  Now we need to rotate right on 4  =======================  Rotate right on 4  9  3(1,0) 10 (0,0)  2(0,0) 4(0,0) |

|  |  |  |
| --- | --- | --- |
| Rotate left  2 (x)  3 (Y)  NULL(T2)  Rotate right    4(Y)  3(x)  2 NULL(T2) | 3  2  NULL  3  2 4 | node \*y = x->right;  node \*T2 = y->left;  // Perform rotation  y->left = x;  x->right = T2;  node \*x = y->left;  node \*T2 = x->right;  // Perform rotation  x->right = y;  y->left = T2; |