AVL Trees

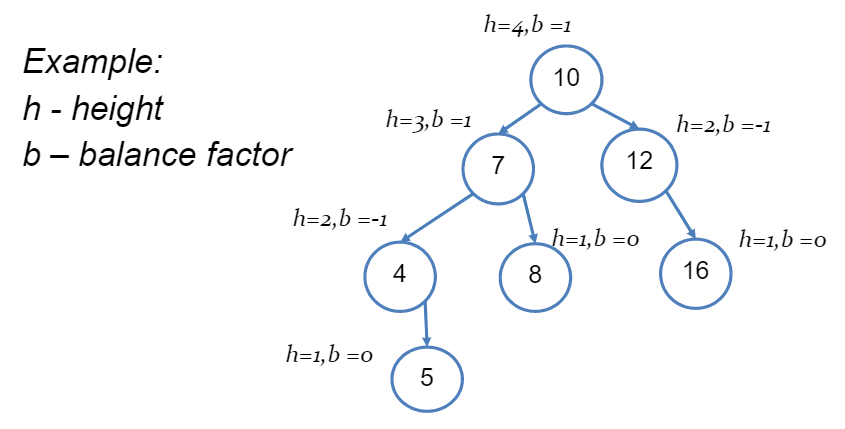
AVL trees are balanced BSTs with height O(log n). AVL search, insert and deletion operations can be done in O(log n) time and O(1) space.

**Concept of a balanced tree**

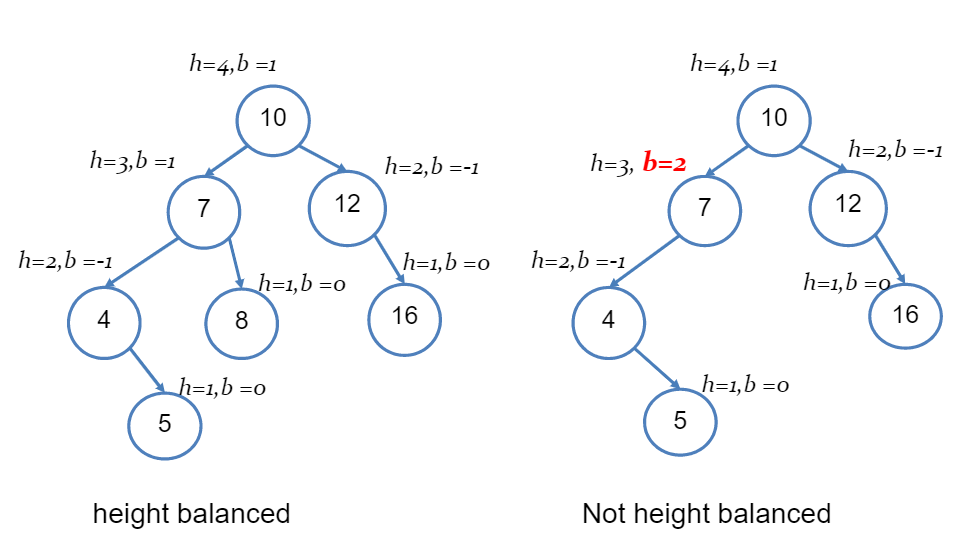
* Balanced binary tree, i.e. height of left and right subtrees are equal or do not have many different at any node
* Ex. A perfect binary tree of n nodes is a balanced tree. The height of a perfect binary tree is O(log2n). The search in perfect BST can be done in O(log2n) time and O(1) space.
* Note that in general a BST is not balanced

**Balance factor**

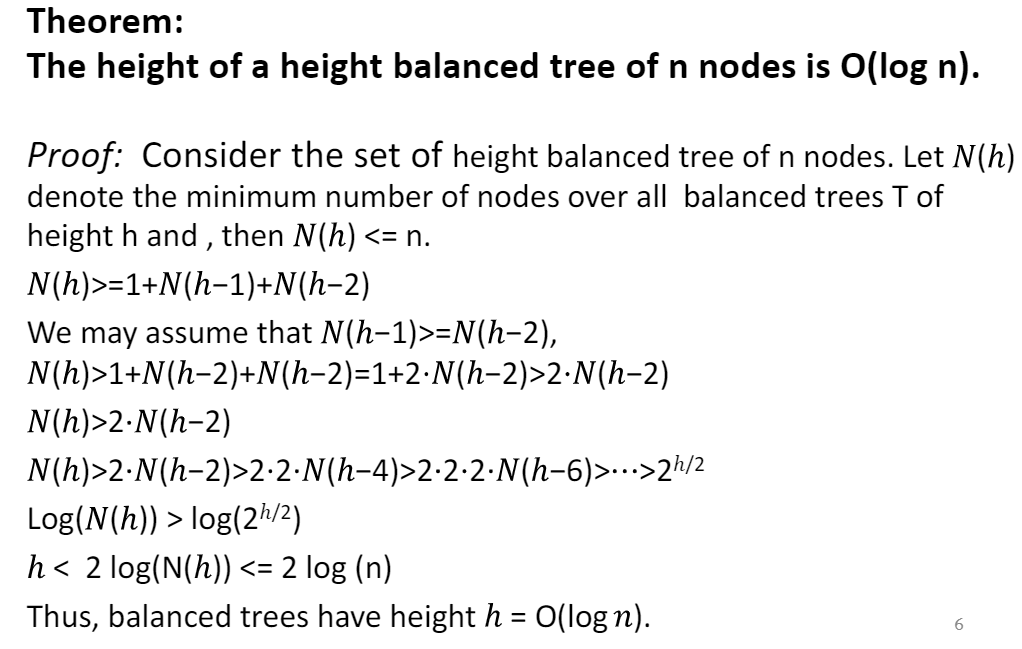
* The balance factor of a node is defined as the heigh of its left subtree minus the height of its right subtree



* A node is left heavy if its balance factor is bigger than 0
* A node is right heavy if its balance factor is less than 0
* Is node is balanced if its balance factor is -1, 0, or 1
  + Otherwise it is unbalanced
* A binary tree is said to be height balanced if all its nodes are balanced



**Height balanced tree theorem**



**AVL trees**

* An AVL tree is a height balanced binary search tree with self-balancing insert and delete operations
* Self balancing operation:
  + Do BST insert or delete
  + Do AVL rotations to restore the height balance property of the BST of step 1
* AVL is named by its inventor Adelson-Velskii and Landis

**Properties of AVL trees**

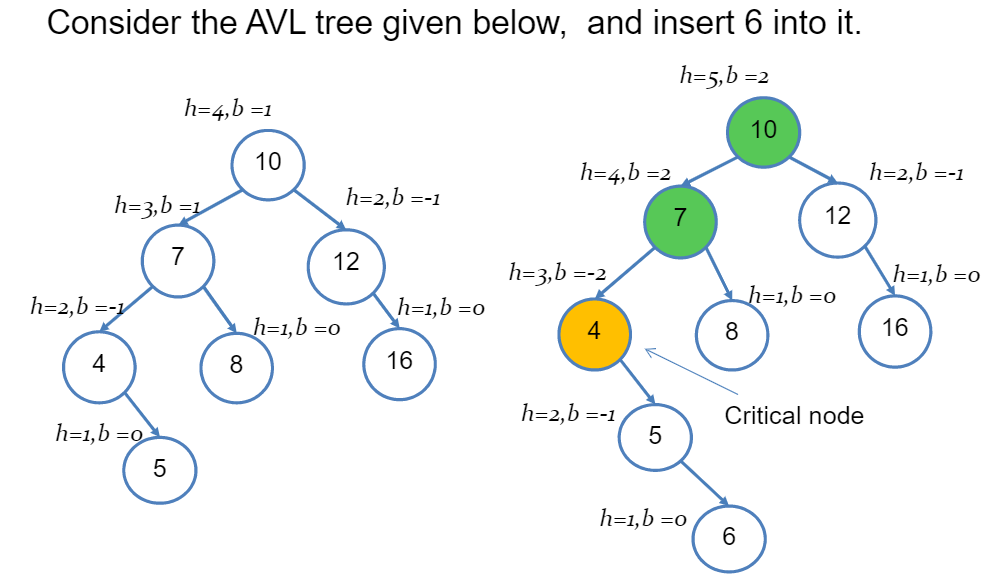
* The key advantage of using an AVL tree is that it takes O(log n) time to perform search, insert, and delete operations
* The space complexity of AVL tree search, insert and delete operation is O(log n) if implemented by recursion algorithm, and O(1) if implemented by iterative algorithm
* As a trade-off for the O(log n) time performance, an additional variable is used to hold the height or balance factor of a node in node structure of AVL trees

**AVL tree insertion**

Algorithm:

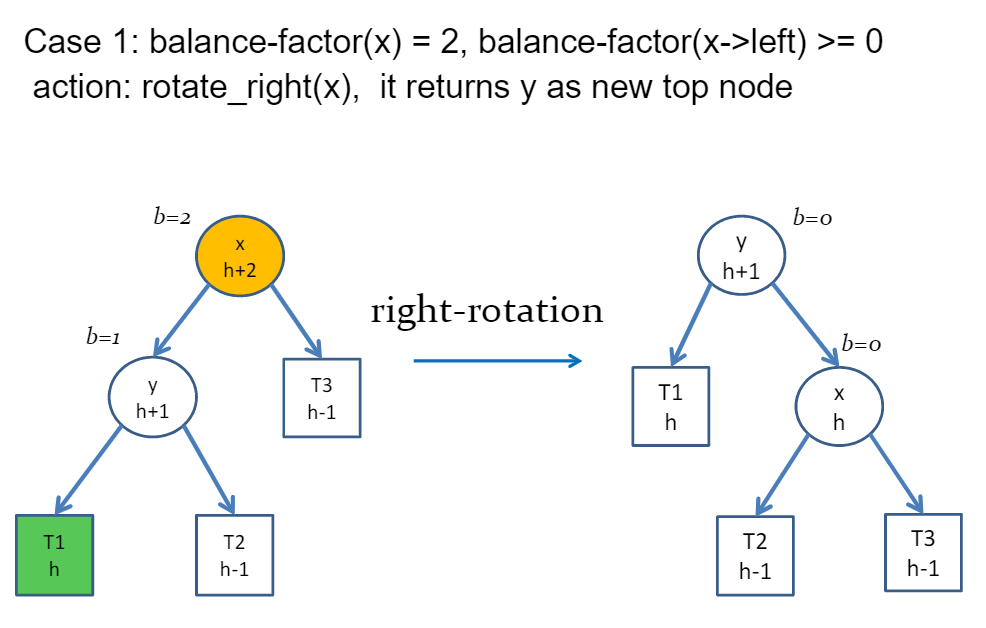
1. Insert node
2. Re-balance to derive a height balanced binary tree

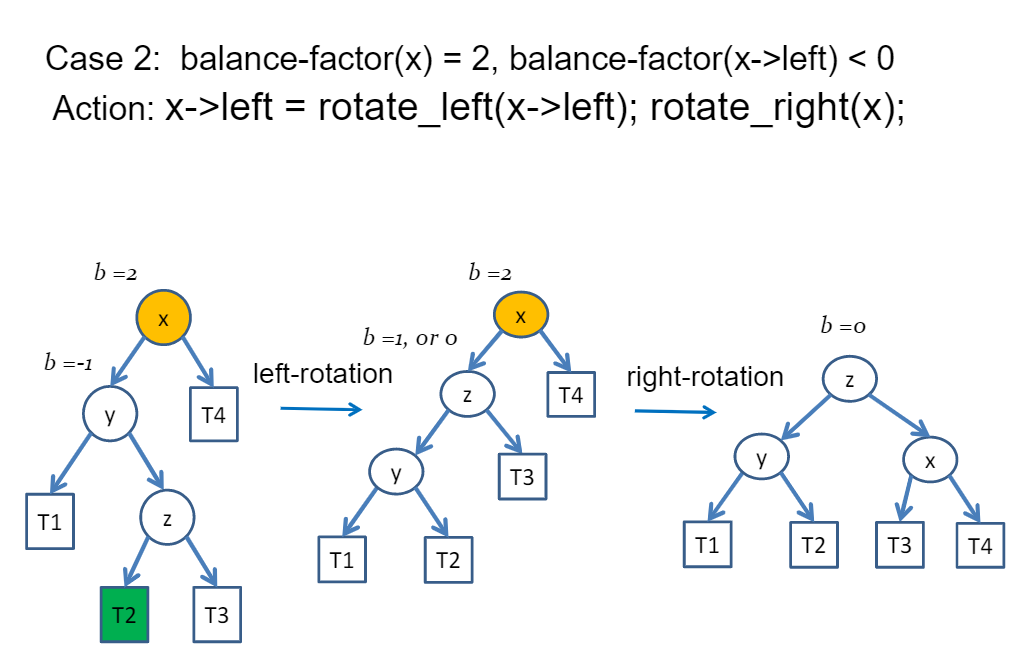
* In step 1, a new node is inserted at the leaf node. The balance factor of the new node is 0. The nodes whose heights will possibly increase by 1 are those who lie on the path from the root to the new node. The changes of heights of the nodes will possibly change the balance factors, thus created unbalanced nodes on the path. The unbalanced node with the smallest height (i.e. of the biggest depth, or the first unbalanced node on the path from the new node to root) is called the *critical node*.

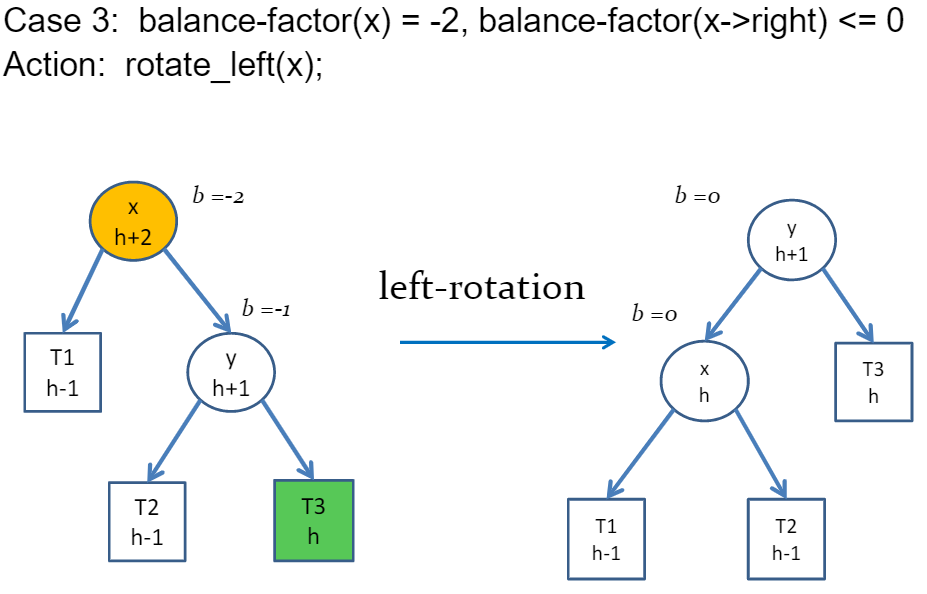


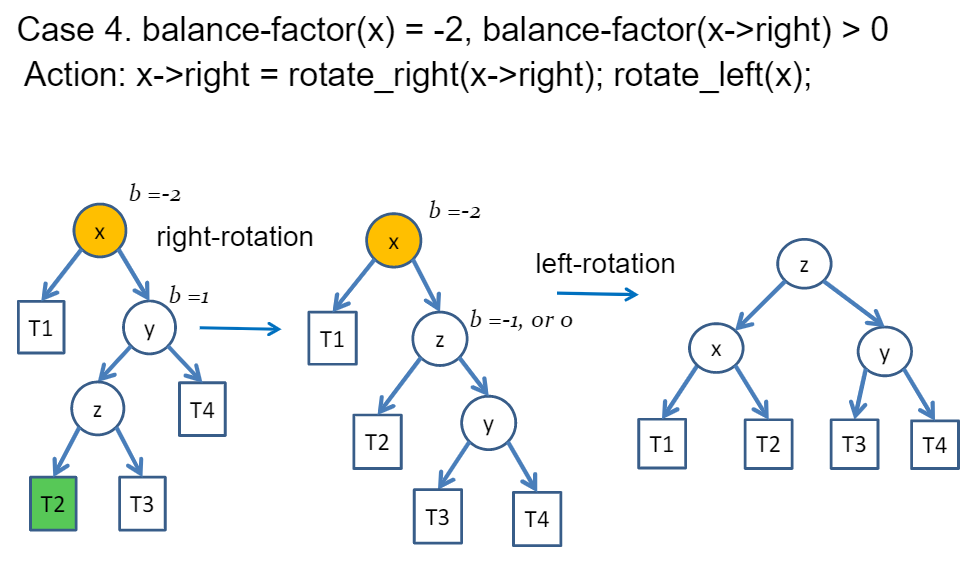
**Critical node patterns and actions**

* Height of the critical node increases by 1, the balance factor of the critical node increases by 1 or decreases by 1, so the balance factor of the critical node is either 2 or -2. There are four possible cases (patterns), and actions (rotations) are taken accordingly for re-balancing. Let x denote the critical node.
* Case 1: balance factor(x) = 2, balance factor (x->left)>=0
  + Action: right rotation
* Case 2: balance factor(x) = 2, balance factor (x->left)<0
  + Action: left-right rotation
* Case 3: balance factor(x)=-2, balance factor(x->right) <=0,
  + Action: left-rotation
* Case 4: balance factor(x) = -2, balance factor (x->right)>0
  + Action: right-left rotation









**AVL tree deletion**

Algorithm:

1. Delete node
2. Do AVL rotation to restore height balanced BST

If the derived BST of step 1 is not height-balanced, find the critical node. There are four possible cases similar to the insertion, do corresponding rotations.

**AVL tree implementation**

Typedef struct node{

Int data; //used as key

Int height; // maintained runtime data, used for computing balance factor

Struct node \*left;

Struct node \*right;

}TNODE;

Int max(int x, int y) {return (x>=y)?x:y;}

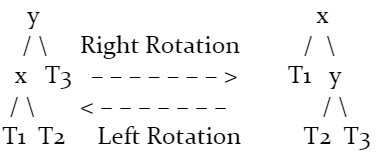
Int height (TNODE \*np) {return (np==NULL)?0:np->height;}

Int balance\_factor(TNODE \*np){

Return (np==NULL)? 0 : height(np->left)-height(np->right);

}

TNODE \*rotate\_right(TNODE \*y){

 TNODE \*x = y->left;

TNODE \*T2 = x->right;

//perform rotation

x->right = y;

y->left = T2;

//update heights

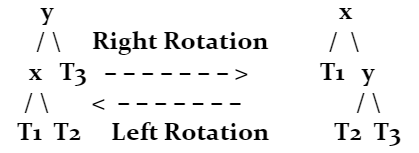
y->height = max(height(y->left), height (y->right))+1;

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

return x;

}

TNODE \*rotate\_left(TNODE \*x){

 TNODE \*y = x->right;

TNODE \*T2 = y->left;

//perform rotation

y->left = x;

x->right = T2;

//update heights

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

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

return y;

}

**Recursive insert algorithm, time O(log n), space O(log n)**

TNODE \*insert (TNODE \*root, int key){

//step 1 do BST insertion

If(root==NULL)

Return new\_node(key);

Else if (key==root->data) //no insertion

Return root;

Else if (key<root->data)

Root->left = insert(root->left, key);

Else //if (key>root->key)

Root->right = insert(root->right, key);

//step 2 re-balancing

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

Int bf = get\_balance\_factor(root);

If(bf==2 && get\_balance\_factor(root->left)>=0)

Return rotate\_right(root);

Else if (bf==2 && get\_balance\_factor(root->left)<0){

Root->left = rotate\_left(root->left);

Return rotate\_right(root);

}

Else if (bf==-2 && get\_balance\_factor(root->right)<=0)

Return rotate\_left(root);

Else if(bf==-2 && get\_balance\_factor(root->right)>0){

Root->right = rotate\_right(root->right);

Return rotate\_left(root);

}

}

**Recursive deletion**

TNODE \*delete(TNODE \*root, int key){

//step 1 do BST deletion

//step 2 re-balancing

//update height of this node

//get balance factor of this node

//do AVL rotations according to balance factor pattern

Return root;

}

Refer to code example 25 for example of iterative algorithm implementation