Solutions to Assignment 5

1. Level order traversal of an arbitrary tree (BFS):

```
levelOrder():
   Queue<Entry> q ← new LinkedList<>()
   q.add(root) // root at depth 0 is explored first
   // LI: depth of nodes in q: {d,d,...,d,d+1,...,d+1}
   while q is not empty do
         ent ← q.remove()
         visit(ent)
         if ent.children != null then
            for c in ent.children do
                  q.add(c)
```

Correctness: At any time of the algorithm, the queue contains 1 or more nodes at depth d, followed by 0 or more nodes at depth d+1 (for some integer d). This is true at the beginning, when the queue contains just the root node, which is at depth d=0. The algorithm works by removing a node from the front of the queue (at depth d), visits it, and places all its children (at depth d+1) at the rear of the queue. Eventually, all nodes at depth d are processed, and the queue contains nodes at depth d+1 only, which is the next value of d. Therefore, the algorithm visits nodes in the order of nondecreasing depth.

If a traversal that visits nodes in order of nonincreasing depth is desired, we can replace “visit(ent)” by stack.push(ent), creating a stack of nodes. At the end, we can pop nodes off the stack and visit them.

RT analysis: The algorithm adds every node to the queue once. Work done in processing a node is proportional to the number of its children. Total work done by the algorithm is $O(|V|+|E|)$, where V is the set of nodes of the tree, and E is the set of its edges. In a tree, $|E| = |V| - 1$. Therefore, the running time of the algorithm is $O(n+n-1) = O(n)$, where $n = |V|$.

Searching a file system: This problem can be solved using any traversal of the tree, where a node is processed for a match when it is visited. We write 2 out of many possible solutions for this problem. RT of either version is O(n).

```
class Pair:
   Entry ent;  String path;
   Pair ( Entry n, String p ) { ent = n; path = p; }

visit ( ent, path ):  // helper method to output entry
   fullName ← path + ent.name
   if ent.isFolder() then
         print fullName + “    folder”
   else
         print fullName + “    file    “ + ent.size

find ( f, pattern ):  // BFS solution
   Queue<Pair> q ← new LinkedList<>()
   q.add( new Pair ( f.root, “/” ) )
   // LI: p in q, p.path has path from root to p.ent
   while q is not empty do
         p ← q.remove()
         n ← p.ent
         path ← p.path
         if  isMatch( n.name, pattern ) then
            visit ( n, path )
         if n.isFolder() and n.contents != null then
            cpath ← path + n.name + ”/”
            for c in n.contents do
                  q.add( new Pair( c, cpath ) )

find ( f, pattern ):  // preorder solution
   find ( f.root, pattern, “/” )

// Precondition: “path” stores path from root to n
find ( n, pattern, path ):  // preorder solution
   if  isMatch ( n.name, pattern ) then
         visit ( n, path )
   if n.isFolder() and n.contents != null then
         cpath ← path + n.name + ”/”
         for c in n.contents do
            find ( c, pattern, cpath )
```

Solutions to some problems on trees:
1. Build BST, given inorder, postorder traversals:

```
build( In, Post ):
   n ← In.length
   tree ← build( In, 0, Post, 0, n )
   return new BST( tree, n )

search( arr, i, s, x ):// find index of x in arr[ i..i+s-1 ]
   // Can be solved in many ways

// Build tree from In[ i..i+n-1 ] and Post[ p..p+n-1 ]
build( In, i, Post, p, n ):
   if n <= 0 then
         return null
   else  // one or more elements
         rootElement ← Post[ p+n-1 ]
         root ← search( In, i, n, rootElement )
         Ln ← root - i   // In[ i..root-1 ]
         Rn ← n - Ln - 1  // In[ root+1.. i+n-1 ]
         left ← build( In, i, Post, p, Ln )
         right ← build( In, root+1, Post, p+Ln, Rn )
         return new Tree( rootElement, left, right )
```

RT of build depends on implementation of search:

Algorithm	**Search RT**	**Build RT**
Linear search	$O(n)$	$O(n^2)$
Concurrent search from both ends	$O(n)$	$O(n \log n)$
Counting sort (if applicable)	$O(1)$	$O(n)$
Hashing	$O(1)$ exp	$O(n)$ exp
BST	$O(\log n)$	$O(n \log n)$

2. Verify validity of a BST.

```
verify( t ): // top down algorithm.  RT = O(n).
      return verify( t.root, −∞, +∞ )

/* check if t is a bst, all of whose elements are
between lb and ub (not inclusive) */
boolean verify( ent, lb, ub ):
   if ent = null then
         return true
   else
         return lb < ent.element and
                  ent.element < ub and
                  verify( ent.left, lb, ent.element ) and
                  verify( ent.right, ent.element, ub)
```

Rewrite the code without using infinity by using null in place of $-\infty$, $+\infty$.

Bottom-up algorithm for same problem.
Recursive method verify() returns a 3-tuple: a boolean indicating whether the tree is a valid bst, its minimum element, and its maximum element. If the tree is not a valid BST, then the min and max elements are arbitrary.

```
verify( t ):  // bottom up algorithm.  RT = O(n)
   if t.size() = 0 then return true
   ( flag, min, max ) ← verify( t.root )
   return flag

(boolean, T, T) verify( ent ):
   cur ← ent.element
   lmin ← cur
   rmax ← cur
   if ent.left != null then
         ( flag, lmin, lmax ) ← verify ( ent.left )
         if not flag or lmax >= cur then
            return ( false, lmin, lmax )
   if ent.right != null then
         ( flag, rmin, rmax ) ← verify ( ent.right )
         if not flag or cur >= rmin then
            return ( false, lmin, rmax )
   return ( true, lmin, rmax )
```

AVL Trees - extension of binary search trees (BST)

AVLTree extends BST.

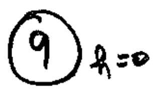
    class Entry extends BST.Entry

        int height;   // height of node in BST

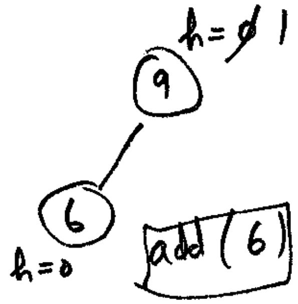
In addition to the ordering conditions of a BST, AVL trees also satisfy the following balance condition at every node of the tree: $|\text{left.height} - \text{right.height}| \leq 1$

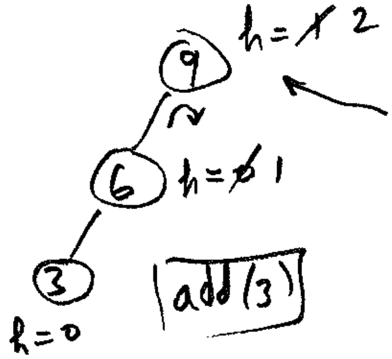
(note: if left = null, use −1 for left.height).

As the tree changes due to add/remove operations, rotations are performed to restore balance.



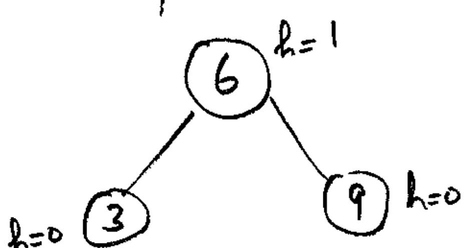
add(9)



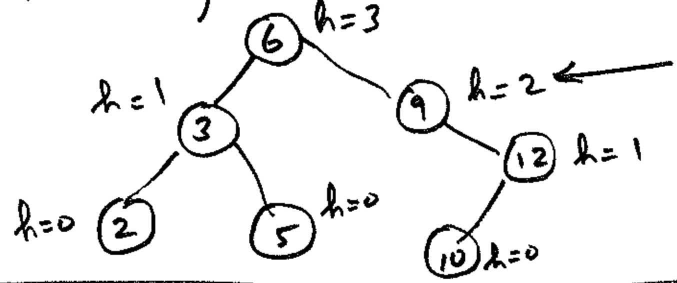


violates balance
left.height = 1
right.height = −1 (null

Rotate tree right around node 9:

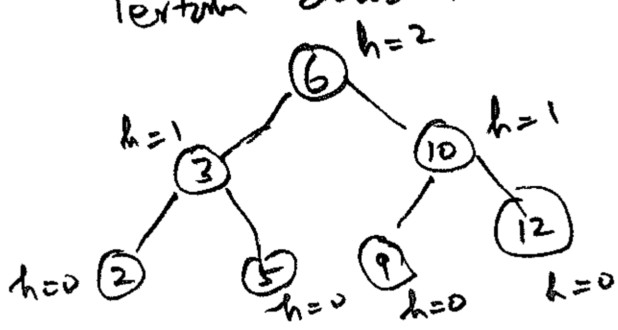


After adding 5, 2, 12, 10 (in that order):

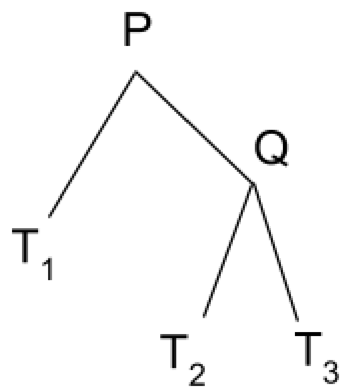
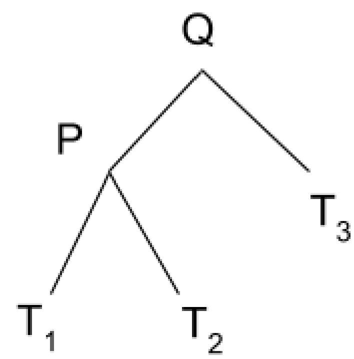
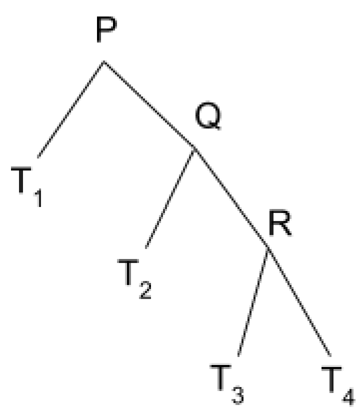
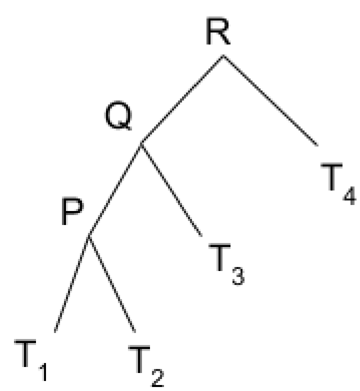
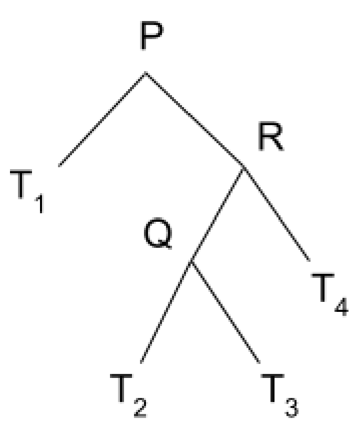
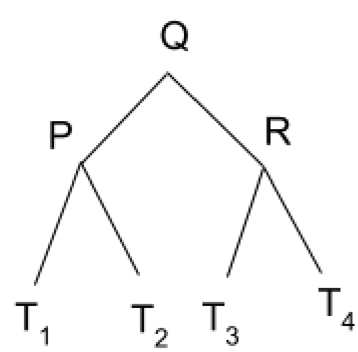


Violates balan
left.height = −
right height = 1

Perform double rotation (Zig-Zag) around 9



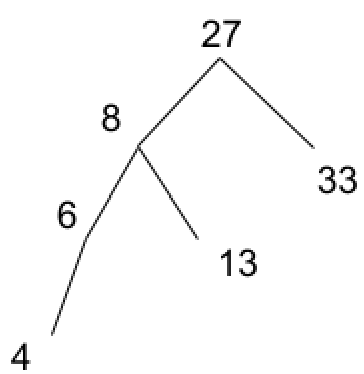
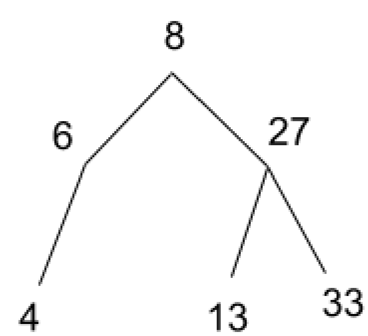
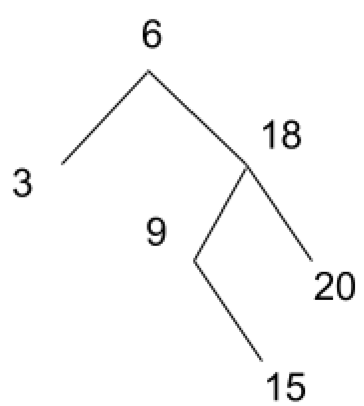
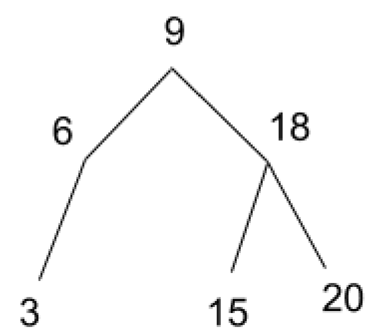
<u>Balanced BST</u>: a BST in which every node satisfies some balancing condition between its left and right subtrees. The goal is to keep the height of trees to be O(log n). When nodes go out of balance, because of add or remove operations, the tree is rotated to restore balance at all nodes. In the following examples, the balance property is violated at node P because of an operation in the subtree of a child or a grandchild.

<table>
<tr>
<td>[R] Single rotation (Zig): Left rotation at P →

</td>
<td>← Right rotation at Q [L]

</td>
</tr>
<tr>
<td>[RR] Double left rotation(Zig-Zig) at P due to R →

</td>
<td>← Double right rotation at R due to P [LL]

</td>
</tr>
<tr>
<td>[RL] Double rotation (Zig-Zag) at P due to Q →

</td>
<td>Tree after rotation:

</td>
</tr>
<tr>
<td>Symmetric case [LR] is not shown: double rotation</td>
<td></td>
</tr>
</table>

Rotation operations used to restore balance in search trees

AVL Tree: A binary search tree that satisfies the following balance condition at every node: the difference between the heights of the left subtree and the right subtree is at most one. A new field is added to Entry class of tree node to keep track of the height of the subtree rooted at that node.

AVL trees inherit all operations of the BST class. When an add operation is performed, the height of ancestors of the new node may increase by 1. Some of these nodes may violate the balance condition. A single or double rotation is performed at the lowest node that goes out of balance, to restore balance to the tree. Similarly, when a remove operation is performed, the heights of ancestors of the removed node may decrease by 1. As a consequence, some nodes may go out of balance. Just one single or double rotation is needed at the lowest node that goes out of balance, to restore balance to the tree.

<table>
<tr>
<td>C1: Left subtree of left child of node u has excess height: rotate right at u. Single right rotation [R] at 27 in example below:

</td>
<td>C1: Tree after rotation is shown below. Heights of nodes involved in the rotation needs to be updated after the rotation (8, 27 in this example):

</td>
</tr>
<tr>
<td>C2: Left subtree of right child of u has excess height. Perform a double rotation [RL] at u. Double rotation [RL] is needed at 6 in this example:

</td>
<td>C2: Tree after rotation is shown below. Heights of nodes 6, 18, and, 9 need to be updated.

</td>
</tr>
<tr>
<td>C3: Right subtree of right child of u has excess height. Symmetric to C1: Single left rotation [L] at u.</td>
<td>C4: Right subtree of left child of u has excess height. Symmetric to C2. Double rotation [LR] at u.</td>
</tr>
</table>

Update operations on AVL trees have a downward pass to a new node that is added or removed, and an upward pass updating the height of nodes of affected ancestors. In all cases, at most one (single or double) rotation is performed to restore balance. Some implementations store left.height − right.height (difference in heights of subtrees) instead of height.