7. BST and AVL

Problem 7.1

Consider the binary search tree shown in Figure 7.1.

1. Draw the tree after inserting the keys: 3, 9, 7 and 4.
2. Draw the tree (obtained in Step 1 after all 4 inserts) after deleting the keys: 1, 10, 8 and 9.

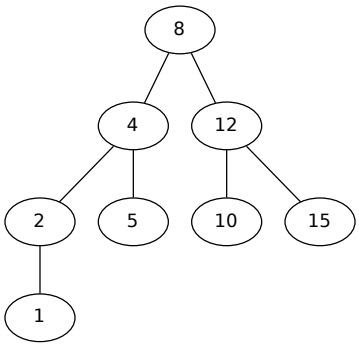


Figure 7.1: A binary search tree (1 is the left child of 2).

Problem 7.2

1. Write a **recursive** version of the method `findKey`, member of the class `BST` (see the specification in the slides).
2. Write the method `removeLarger`, member of the class `BST` that takes as input a key k that exists in the tree and removes all the keys that are (strictly) larger than k. The method signature is `public void removeLarger(int k)`.
3. Write the method `nbLess`, member of the class `BST` that takes as input a key k and returns the number of keys in the tree that are smaller or equal k. The method signature is: `public int nbLess(int k)`.

Problem 7.3

1. List the **keys** (not the data) of the tree shown in Figure 7.3 in **reverse inorder** (right, node, left). What do you notice?
2. Write the method `kLargest`, member of the class `BST` that takes as input an integer k and returns a queue that contains the data corresponding to the k largest keys in the tree. The order of the data in the queue must be in reverse order of that of the keys. Assume that k is less or equal the size of the tree.
   The signature of the method is: `public Queue<T> kLargest(int k)`.

   ■ **Example 7.1** For the tree shown in Figure 7.3, the output to `kLargest(4)` is the queue: $G \to C \to F \to A$. ■

   Hint: Use the observation you made in Part 1 of this problem and combine it with the idea of Problem 2.1.

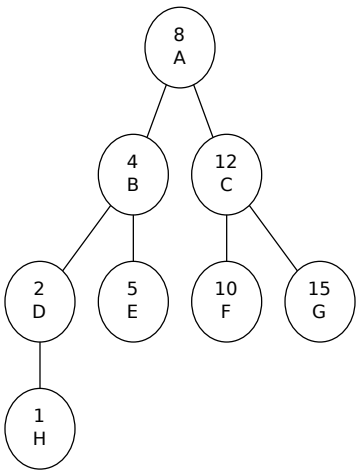


Figure 7.2: A binary search tree (H is the left child of D). The keys are the numbers, whereas the letters are the data.

Problem 7.4

Write the method `intervalSearch`, member of the class `BST`, that takes as input two keys k_1 and k_2 and returns a queue that contains the data of all nodes that have keys in the interval $[k_1, k_2]$. The order of the data in the queue must be the same as that of the keys.

The method signature is: `public <T> intervalSearch(int k1, int k2)`.

■ **Example 7.2** For example, the call to the method `intervalSearch(5,14)` on the tree shown in Figure 7.3, returns the queue that contains: $E \to A \to C \to F$. ■

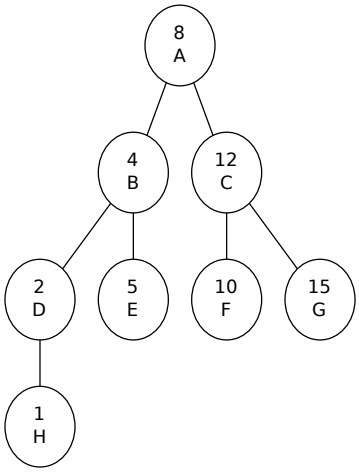


Figure 7.3: A binary search tree (*H* is the left child of *D*). The keys are the numbers, whereas the letters are the data.

Problem 7.5

1. As a member of the class BST, write the method `boolean isLeftRangeLarger(BSTNode<T> t)` that measures the range of possible values that can be found in the left subtree of t and compares it with the range of its right subtree. The method returns true if left is larger, false otherwise.

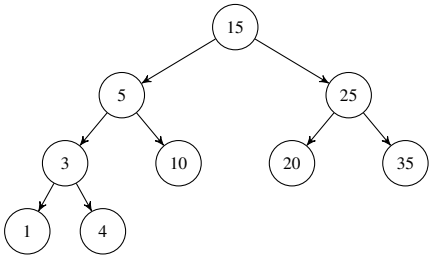


Figure 7.4: Binary Search Tree.

■ **Example 7.3** For the BST in Figure 7.4, `isLeftRangeLarger(root)` will measure the left range as 15-1=14 and the right range as 35-15=20 which means the method will return false.

■

2. Write a method `boolean isSearchPathLinear(int k)` that checks if the search path of the given key k has a linear structure or not. A linear structure means that the path from root to the key k has either all left edges or all right edges (assume that k exists in the tree).

■ **Example 7.4** For the BST in Figure 7.4, if the method receives the key 35, the method `isSearchPathLinear` will return true given the path is all right edges. If the method gets the key 10, the method will return false. ■

Problem 7.6

1. Write the method `secondMin`, member of the class `BST`, that returns the data associated with second minimum key in the tree. Assume that the tress has at least two keys. The method signature is: `private T secondMin(BSTNode<T> t)`.
2. Write the method `inSubtree`, member of class `BST`, that will check if a given key k_2 is in the subtree of another given key k_1. The method should return true if k_1 exists and k_2 is in the subtree rooted at k_1, false otherwise. The method signature is `public boolean inSubtree(int k1, int k2)`.

Problem 7.7

1. Write the method `private void swapData(int k)`, member of the class `BST`, that swaps the data associated with the key k with the data of its parent. If the key k does not exist or the corresponding node has no parent, the tree is unchanged. **Do not call any methods of the class `BST`.**
2. Write a member method that prints the keys of a BST in reverse order (note that in-order traversal of a BST visits the nodes in increasing order, so how can we visit the largest keys first?).

Problem 7.8

1. Write the method `public int nbInRange(int k1, int k2)`, member of the class `BST`, which returns the number of keys k in the `BST` that statisfy: $k_1 \leq k \leq k_2$. Make sure that the method does not visit any unncessary nodes.
2. Write the method `private int deepestKey(BSTNode<T> t)`, member of `BST` which returns the key associated with the deepest node in the subtree t. In case of a tie (two keys at the same depth), the smallest key should be returned. As precondition, t must not be empty.
3. Write the member method `public int maxKey (int k)` of the class BST (binary search tree) that returns the maximum key of the sub-tree rooted at the node with key k. Assume that k exists.

■ **Example 7.5** For the tree below, `maxKey(16)` returns 18, `maxKey(48)` returns 48.

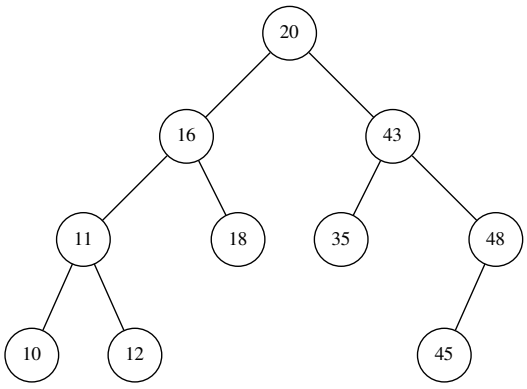


Figure 7.5: BST example

■

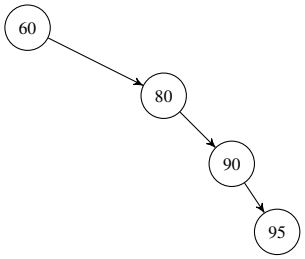


Figure 7.6: BST Tree.

Problem 7.9

Given the BST tree in Figure 7.6, what are the needed steps to convert this BST into an AVL tree? In English, write the algorithmic steps needed to do the task. Draw the AVL tree after converting the BST. Make sure to show your steps clearly including: what type of rotation is performed (rotate left or right) and show the balance of the tree after each step.

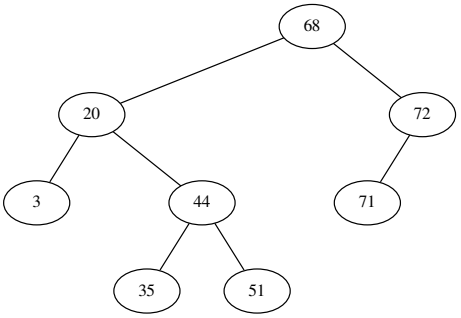
Problem 7.10

Consider a BST tree with nodes containing a pointer to the parent node.

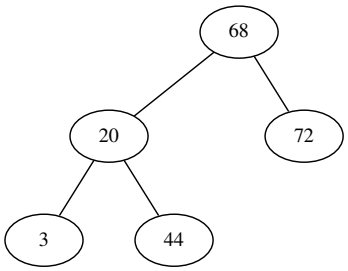
1. Write the **non-recursive** methods: `BSTNode<T> nextPreorder(BSTNode<T> t)`, `BSTNode<T> nextInorder(BSTNode<T> t)`, `BSTNode<T> nextPostorder(BSTNode<T> t)`, which return the node following `t` in the specified order.
2. Show that all three traversals visit the leaf nodes if the same order.
3. Use one of these methods to find the next right child leaf following a leaf node `t`.

Problem 7.11

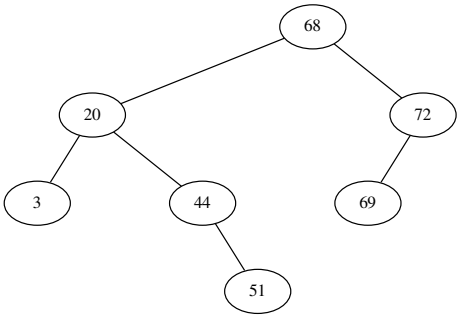
1. Draw the following AVL trees after inserting each of the specified keys. Indicate the pivot if it exists and the type of rotation used to balance the tree if required.



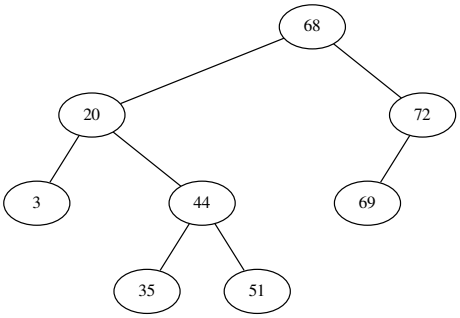
(a) Insert 69



(b) Insert 48

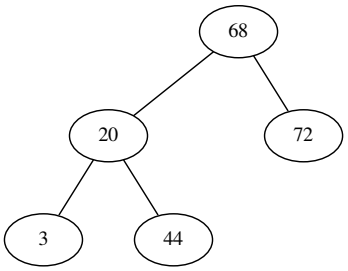


(c) Insert 70

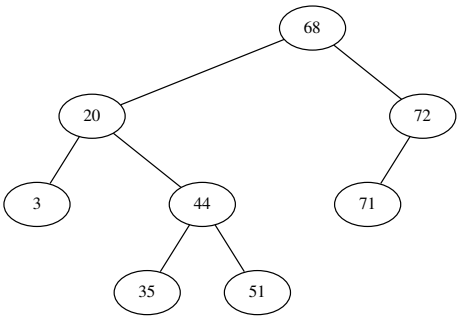


(d) Insert 38

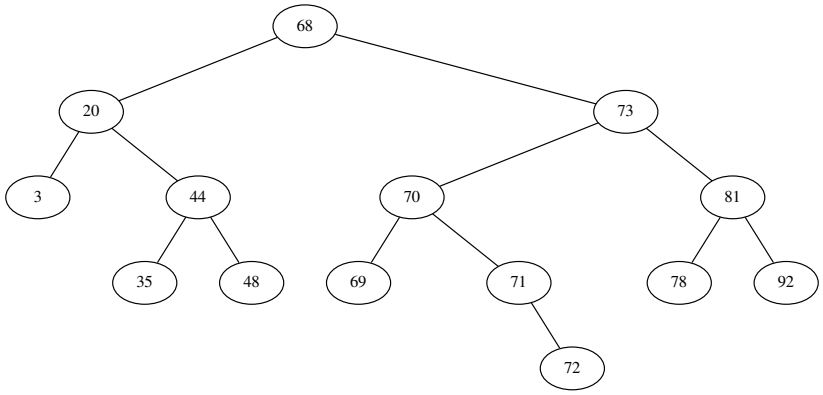
2. Draw the following AVL trees after deleting each of the specified keys. Indicate the type of the rotation(s) used to balance the tree if required.



(a) Delete 72



(b) Delete 71



(c) Delete 3

Problem 7.12

Complete the table below by computing the resulting balance for each of the specified rotations made on the trees shown in Figure 7.7.

Case	Initial balance	Rotation	Resulting balance
(a)	Bal(A)=-2, Bal(B)=-1	R(A)	Bal(A)=?, Bal(B)=?
(a)	Bal(A)=-2, Bal(B)=0	R(A)	Bal(A)=?, Bal(B)=?
(b)	Bal(A)=2, Bal(B)=1	L(A)	Bal(A)=?, Bal(B)=?
(b)	Bal(A)=2, Bal(B)=2	L(A)	Bal(A)=?, Bal(B)=?
(c)	Bal(A)=-2, Bal(B)=1, Bal(c)=-1	L(B), R(A)	Bal(A)=?, Bal(B)=?, Bal(C)=?
(c)	Bal(A)=-2, Bal(B)=1, Bal(c)=0	L(B), R(A)	Bal(A)=?, Bal(B)=?, Bal(C)=?
(c)	Bal(A)=-2, Bal(B)=2, Bal(c)=-1	L(B), R(A)	Bal(A)=?, Bal(B)=?, Bal(C)=?
(d)	Bal(A)=2, Bal(B)=-1, Bal(c)=0	R(B), L(A)	Bal(A)=?, Bal(B)=?, Bal(C)=?
(d)	Bal(A)=2, Bal(B)=-1, Bal(c)=-2	R(B), L(A)	Bal(A)=?, Bal(B)=?, Bal(C)=?

■ **Example 7.6** For the first line:

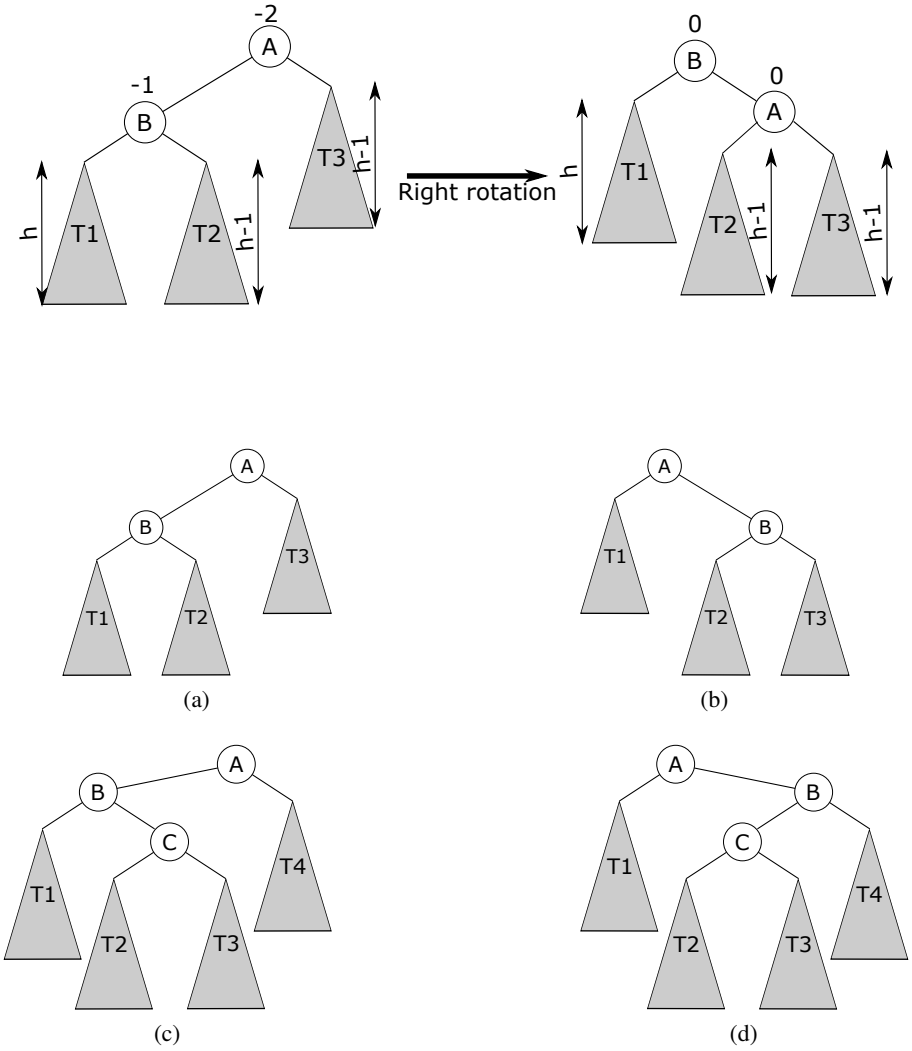


Figure 7.7: Trees.

Problem 7.13

1. Write:
   (a) The method `private AVLNode<T> rRot(AVLNode<T> p)`, member of the class `AVLTree` that performs a right rotation at p and returns the new root of the subtree.
   (b) The method `private AVLNode<T> lRot(AVLNode<T> p)`, member of the class `AVLTree` that performs a left rotation at p and returns the new root of the subtree.
   (c) Use the two previous methods to write the method `private AVLNode<T> rlRot(AVLNode<T> p)`, member of the class `AVLTree` that performs a right-left rotation at p and returns the new root of the subtree.
2. Write the method `private int height(AVLNode<T> t)`, member of the class `AVLTree`, that returns the height h of the subtree rooted at t (an empty subtree has height 0). The methods must be $O(h)$ in time.

Problem 7.14

★ The goal of this problem is to write an algorithm for updating balance after a single rotation for arbitrary initial balances.

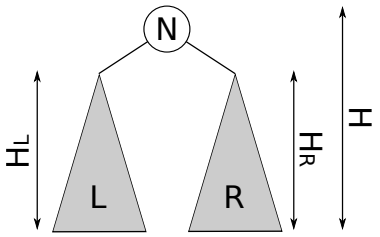
1. Consider the two functions *pos* and *neg* defined as follows:

$$pos(n) = \begin{cases} 0 & \text{if } n < 0, \\ n & \text{if } n \geq 0. \end{cases} \tag{7.1}$$

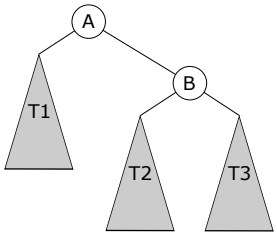
$$neg(n) = \begin{cases} 0 & \text{if } n > 0, \\ -n & \text{if } n \leq 0. \end{cases} \tag{7.2}$$

   Demonstrate the following properties:
   (a) $pos(n) = \dfrac{|n|+n}{2}$.
   (b) $neg(n) = \dfrac{|n|-n}{2}$.
   (c) $pos(n) + neg(n) = |n|$.
   (d) $pos(n) - neg(n) = n$.
2. Consider the general case of a binary tree as shown below. Show that:

$$H = H_L + pos(bal(N)) + 1 = H_R + neg(bal(N)) + 1. \tag{7.3}$$



3. Consider the tree below after a left rotation at A. Let a, b be the balance of A and B respectively before rotation, and a', b' their balance after rotation.



   (a) Show that:

$$\begin{cases} a' = a - pos(b) - 1, \\ b' = b - neg(a') - 1. \end{cases} \tag{7.4}$$

   (b) Use this result to write the method `private AVLNode<T> lRot(AVLNode<T> t)` for left rotation of the sub-tree `t`. The method updates the balance of A and B, and returns the new sub-tree obtained after rotation.
4. Repeat the previous exercise for the case of a right rotation.