**BST Lab**

**Solam Jung Rana(986888)**

1. Draw the BST obtained by inserting the following integers (using successive calls to insert): 1, 9, 3, 8, 12, 4, 2

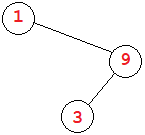
**Insert 1:**

****

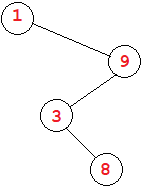
**Insert 9:**

****

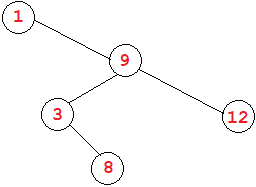
**Insert 3:**

****

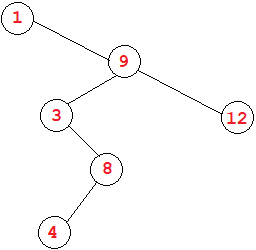
**Insert 8:**

****

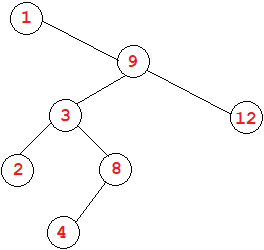
**Insert 12:**

****

**Insert 4:**

****

**Insert 2:**

****

1. Create a sorting routine based on a BST and place it in the sorting environment, distributed earlier. For this, your new class, BSTSort, should be a subclass of Sorter. Your BSTSort class can be essentially the same as the BST class given in the slides (see the folder in your labs directory for this lab), except that you will need to modify the printTree method so that it outputs values to an array (rather than printing to console).

After you have implemented, discuss the asymptotic running time of your new sorting algorithm. Run an empirical test in the sorting environment and explain where BSTSort fits in with the other sorting routines (which algorithms is it faster than? which is it slower than?).

**See BSTSort.java**

**Asymptotic running time: O(log n)**

**MergeSort runs faster, BSTSort has a storage creation and traversal overhead.**

1. For each integer *n* = 1, 2, 3,…, 7, determine whether there exists a red-black tree having exactly *n* nodes, with *all of them black.* Fill out the chart below to tabulate the results:

|  |  |
| --- | --- |
| **Num nodes *n*** | **Does there exist a red-black tree with *n* nodes, all of which are**  **black?** |
| **1** | **YES** |
| **2** | **YES** |
| **3** | **YES** |
| **4** | **NO** |
| **5** | **YES** |
| **6** | **YES** |
| **7** | **YES** |

1. For each integer *n* = 1,2,3,…, 7, determine whether there exists a red-black tree having exactly n nodes, where *exactly one of the nodes is red.* Fill out the chart below to tabulate the results:

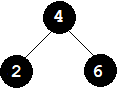
|  |  |
| --- | --- |
| **Num nodes *n*** | **Does there exist a red-black tree**  **with *n* nodes, where exactly *one* of**  **the nodes is red?** |
| **1** | **NO** |
| **2** | **YES** |
| **3** | **NO** |
| **4** | **YES** |
| **5** | **YES** |
| **6** | **NO** |
| **7** | **NO** |

**Lab 10**

1. A red-black tree is said to be *derivable* if it is obtained from an insertion sequence of nodes, using the rules for insertions starting from an empty tree. Give an example to show that not every red-black tree is derivable. (In other words, you can build a BST that satisfies the four conditions for a red-black tree, and yet there is no way to obtain this tree by successively inserting nodes using the insertion algorithm rules.)

**Using the example of insertion of {2, 4, 6}**

**Attempt deriving such a red-black tree:**

****

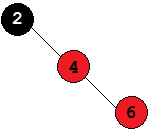
**Insert 2:**

****

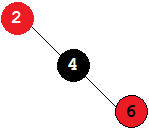
**Insert 4:**

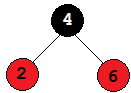
****

**Insert 6:**

****

**Re-structure:**

****

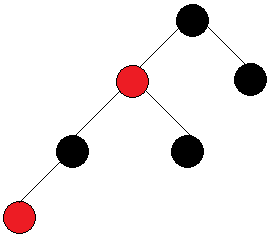
****

**This shows that after performing the insertion algorithm steps, it is not possible to achieve certain types of red-black trees.**

1. An *AVL Tree* is a BST that satisfies a different balance condition, namely:

The AVL Balance Condition For each internal node x, the height of the left child of x differs from the height of the right child of x by at most 1. (Equivalently, the heights of the left and right subtrees of x differ by at most 1.)

Create a red-black tree that does *not* satisfy the AVL Balance Condition.

****

1. Use the insertion algorithm for red-black trees to successively insert the following nodes, starting with an empty tree.

Note on Part (a): Recall that an already sorted insertion sequence is a worst case for an ordinary BST. Notice how the red-black balancing operations handle this to remain balanced.

1. 1, 2, 3, 4, 5, 6, 7, 8

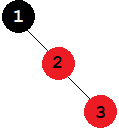
**Insert 1:**

****

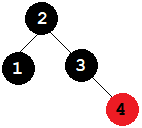
**Insert 2:**

****

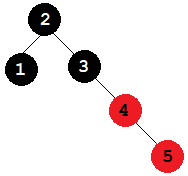
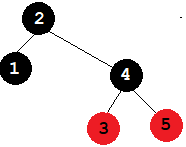
**Insert 3:**

** **

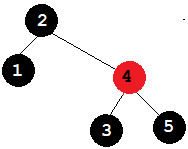
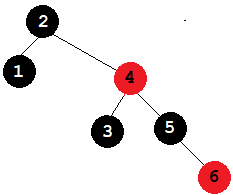
**Insert 4:**

** **

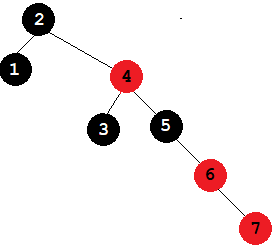
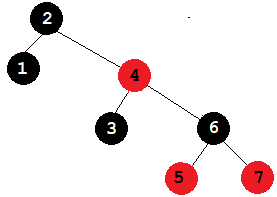
**Insert 5:**

** **

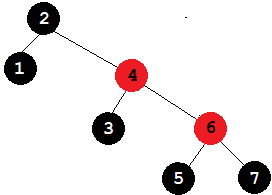
**Insert 6:**

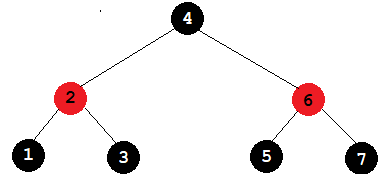
** **

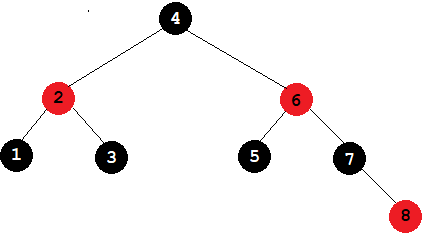
**Insert 7:**

** **

**Insert 8:**

****

****

****

1. 3, 2, 1, 4, 5, 6

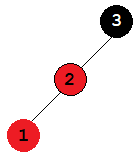
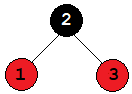
**Insert 3:**

****

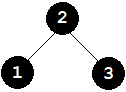
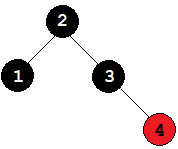
**Insert 2:**

****

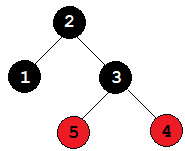
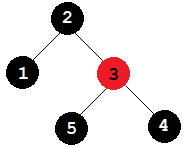
**Insert 1:**

** **

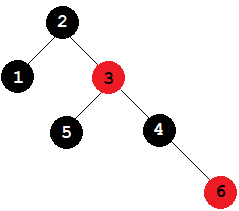
**Insert 4:**

** **

**Insert 5:**

** **

**Insert 6:**

****

1. Devise an algorithm IsPrime(*n*) which outputs TRUE if n is prime, FALSE otherwise. Then implement as a Java method. What is the asymptotic running time of IsPrime?

Explain.

**See PrimeChecker.java**

**Algorithm IsPrime(n)**

**Input: Non-negative integer n**

**Output: TRUE if n is prime, FALSE otherwise**

**if n ≤ 1 then**

**return false**

**for i ← 2 to i \* i ≤ n do**

**if n % i = 0 then**

**return false**

**return true**