**Algorithm: Lab8 (By Sujiv Shrestha ID:610145)**

**Problem 1.**

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.

Solution:

null

null

null

null

null

null

null

For the root node of this Red-Black tree the height of the left child is 2 while the height of the right child is 0. So, this Red-Black tree does not satisfy the AVL Balance Condition.

**Problem2**

2. Use the insertion algorithm for red-black trees to successively insert the following

nodes, starting with an empty tree.

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

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

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.

Yes, it somehow tries to restructure the unbalanced BST to balanced even in the worst case.

**Problem2**

3. 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.

**Algorithm** isPrime(n)

**Input** integer n to check if it is prime

**Output** Boolean: true if it is prime number and false if it is not

**for** i←2 to √n **do**

**if**(n%i=0) **then**

**return** **false**

**return** **true**

Java Implementation:

**public** **static** **boolean** isPrime(**int** n) {

**for**(**int** i=2;i<=Math.*sqrt*(n);i++) {

**if**(n%i==0) {

//Divisible by i

**return** **false**;

}

}

**return** **true**;

}

Running Time, T(n) = O(√n)

4. In the course, we have determined asymptotic running times of sorting algorithms as a function of input size *n.* However, in number-theoretic algorithms, such as GCD, the running time has been bounded by functions of *n* where, in this case, *n* is an input *value*, but does not represent the input *size.* The reason is that the *size* of a natural number *n*, from the point of view of any reasonable computational model, is *its length as a bit string* and not simply the value *n* itself.

Examples:

If n = 7, its size as the bit string 111 is 3.

If n = 67, its size as the bit string 1000011 is 7.

In general, *length*(*n*) = log *n* + 1.

When running times of number-theoretic algorithms are expressed in terms of input *size* rather than input *value* (as we have done so far), results can appear unfamiliar.

For example, the asymptotic running time of GCD(*m,n*) in terms of input values, as we have seen, is O(log *n*). However, since *n* is O(2*length*(*n*) ), in terms of input *size*, GCD(*m,n*) runs in O(*length*(*n*)). That is, GCD(*m,n*) is *linear in the size of n.* Here is a more careful analysis:

**GCD Algorithm**

**Algorithm** GCD(m,n)

**Input** nonnegative integers m, n, not both 0

**Output** gcd(m,n)

**if** n=0 **then**

**return** m

**else**

**return** GCD(n, m % n)

(\*\*) In general, if T(n) is O(f(n)), in terms of the *value n,* then, in terms of the size b = b(n) of input n, T(b) is O(f(2b)). In the case of GCD, since gcd(m,n) runs in O(log n), in terms of the value n, gcd runs in O(log 2b) = O(b) in terms of input size.

In light of the above discussion, answer the following:

A. Express the asymptotic running time of your algorithm IsPrime(*n*) in terms of the input *size* rather than input value. It may be helpful to use two arguments, *n*, *b*(*n*), to help focus on the number of bits of n when computing running time; then you can compute running time *T*(*b*) in terms of the input size. Or you can simply compute the running time in terms of n, then convert to running time in terms of b using the formula given in (\*\*) above.

Solution:

Running Time of my isPrime algorithm, T(n) = O(√n)

The size of the bit stream representing number n is, b = length(n) = ⎣logn⎦+1

So,

T(n) = O(length(√n))

= O(⎣log(√n)⎦+1)

= O(⎣log(n1/2)⎦+1)

= O(⎣1/2\*log(n) ⎦+1)

≅ O(logn)

Or, we can also say that T(b) ≅ O(b)

B. Suppose *T*(*b*) is the running time of your algorithm in terms of input size. Show that *b2* is o(*T*(*b*)).

**Problem3**

3. 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 | No |
| 3 | Yes |
| 4 | No |
| 5 | No |
| 6 | No |
| 7 | Yes |

Red-Black Tree examples:

|  |  |
| --- | --- |
| n=1  null  null | n=3  null  null  null  null |
| n=7  null  null  null  null  null  null  null  null |  |

**Problem4**

4. 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, exactly one of the nodes is red?** | |
| 1 | No |
| 2 | Yes |
| 3 | No |
| 4 | Yes |
| 5 | No |
| 6 | No |
| 7 | No |

|  |  |
| --- | --- |
| n=2, r=1  null  null  null | n=4, r=1  null  null  null  null  null |